Rochester Institute of Technology

RIT Scholar Works

Theses

5-1-2001

# Adaptive virtual protocol stacks for intrusion detection applications

David McGann

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

McGann, David, "Adaptive virtual protocol stacks for intrusion detection applications" (2001). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

# ADAPTIVE VIRTUAL PROTOCOL STACKS FOR INTRUSION DETECTION APPLICATIONS

by

David D. McGann

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Engineering

Department of Computer Engineering
Rochester Institute of Technology
Rochester, New York

May, 2001

Approved by _____
Dr. Yoonhee Kim - Thesis Advisor

_____
Dr. Roy Czernikowski - Thesis Committee Member

_____
Dr. James Heliotis - Thesis Committee Member

Release Permission Form

Rochester Institute of Technology

Adaptive Virtual Protocol Stacks for Intrusion Detection Applications

I, David D. McGann, hereby grant permission to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

David D. McGann

5/13/2001
Date

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

The author wishes to thank his the following Professors at the Rochester Institute of Technology: Dr. Andreas Savakis (Computer Engineering Department Head), Dr. Yoonhee Kim (Thesis Advisor), Dr. Roy Czernikowski (Thesis Committee Member), and Dr. James Heliotis (Thesis Committee Member). Without your dedication and support, this paper would not have been possible. Thank you.

# GLOSSARY

ACK FLAG. Indicates that the value in the acknowledgement number field is valid.

ADAPTIVE CONFIGURATION. The capability to identify changes in the environment and changes the system configuration to compensate.

BINARY FLAGS. Flags that use a binary bit to represent some information.

BLACK BOX TESTING. A testing methodology where only the inputs and outputs of the systems can be examined.

BRO. A network-based intrusion detection system designed to handle high throughput environments by attempting to consolidate information at the lowest possible level.

CERT/CC. CERT coordination center is a site for reporting and alerting the Internet community of security issues. http://www.cert.org

CLIENT DATA SINK. Mechanism used to retrieve results of a test on a target machine.

DENIAL OF SERVICE ATTACK. Prevents the use of a service or network by flooding it with packets.

EVASION ATTACK. Formats network packets by removing characters so that the intrusion detection system will receive the transmitted message differently than the target machine.

FIN FLAG. Used to gracefully terminate a connection.

FRAGMENTATION OFFSET. Determines where fragments fit in reference to the beginning of the original unfragmented packet.

FTPD. An application which provides the file transfer protocol service to users. FTP is used to transfer files across a network.

HETEROGENEOUS ENVIRONMENTS. A collection of computer systems with different architectures and/or operating systems.

HIGH PERFORMANCE NETWORK INTRUSION DETECTION SYSTEM. A rule based pattern matching intrusion detection system. Rules are written in the scripting language provided.

INSERTION ATTACK. Formats network packets by adding characters so that the intrusion detection system will receive the transmitted message differently than the target machine.

IP HEADER. The preamble to an Internet protocol packet that contains information such as source and destination addresses of the packet.

IP PROCESSOR. The part of virtual protocol stacks that manages the Internet protocol identically to the target machine.

LIBCAP LIBRARIES. Public domain software for retrieving raw packets from the network.

MOBILE CODE PLATFORM. A network server application that enables applications to be transmitted by the remote host and executed.

NETWORK DRIVER. Extracts packets from the local network and passes them on to the packet router.

NETWORK INTERFACE. Physical connection to the network such as IEEE 802.3 Ethernet.

PACKET ROUTER. Determines the IP processor that will handle the newly received packet.

PACKET FRAGMENTATION. Divides larger packets into smaller packets that can be transmitted over heterogeneous technologies.

PACKET SNIFFER. Software application that allows a user to examine packets traveling on the network.

PUSH FLAG. Used by the transmitter to inform the receiver to push the data through the buffers as quickly as possible.

RST FLAG. Used to immediately reset a TCP connection.

SNORT. By using a promiscuous network interface device to sniff packets of the local network, it runs a pattern-matching algorithm to determine if any data stream or network packet headers are similar to any known attack.

SYN FLAG. Used to establish a connection during a three-way handshake.

TCP CHECKSUM. Used for error detection of the TCP segment, therefore protecting the header and data.

TCP HEADER. Preamble used to route packets to the proper application and provide reliable in-order delivery.

TCP PROCESSOR. Reassembles the application data stream following the TCP standard and maintains the state of the connection.

TCP SEQUENCE NUMBERS. To send a reliable message, TCP uses sequence numbers to keep track of each byte sent over the link.

TESTER. Evaluates a target machine's TCP/IP implementation, as well as constructs the TCP and IP processors used by virtual protocol stacks.

TIME TO LIVE. Used by the network to limit the amount of time that a packet can bounce around the network.

URG FLAG. Used to inform the receiver that urgent data exists in the packet.

VIRTUAL PROTOCOL STACK. Works with intrusion detection systems to determine how target machines handle anomalous packets and identify all attacks taking advantage of these packet processing inconsistencies.

*Chapter 1*

# INTRODUCTION

## 1.1 Background

As the number of computers connected to the Internet multiplies, computers must be secured against unauthorized access and abuse. Unauthorized access can occur when an external attacker enters the system through a bug in the application's server software or an internal user pretends to be another user. When an internal user attempts to exceed the privileges given by the system administrator, this is considered abuse. Host and network based intrusion detection systems protect computers by monitoring access and actions taken by users.

Two main sources of threats exist in computer systems from internal users and external intruders. Attacks launched by local users are normally attempts to gain more privilege, such as reading private files or gaining access to secluded computers. The internal offender will not necessarily be attacking across a network, which makes a network-based intrusion detection system useless.

External intruders enter through the network to attack a computer system. The first line of defense is the network firewall. However, once penetrated, the network-based intrusion detection system must step in. Intruders may enter by obtaining passwords of authorized users and striving to gain higher privileges from the unsuspecting user's account.

On the other hand, they could also exploit a bug in network service's software, such as the FTPD input validation problems reported by CERT and in return, can execute arbitrary commands with administrative privileges [8]. External intruders violate computers for various reasons, such as looking for company secrets, searching for a launching point for future attacks, or simple mischief. To construct a launching point, intruders install backdoors which allow access to computers without being monitored or logged.

For network-based intrusions, attackers usually enter through multiple launching points for concealment. In order for a system administrator to track down an intruder, he/she must determine where the attack is actually originating. Assuming the attacker has not spoofed an IP address, the system administrator must contact the system administrator of the attacker's machine. This administrator then must determine the source of the intrusion from his/her network. If the attacker has traveled through multiple systems, this step must be repeated at every system. As the Internet grows and international boundaries are crossed, attacking becomes easier, while tracking becomes more difficult.

Instead of using the violated system as a launching point for future intrusions, an attacker could use the system as a client in a distributed denial of service attack. The denial of service attack attempts to prevent the use of a service or network by flooding it with packets. To make this attack successful, attackers will attempt to recruit as many systems as possible, especially ones with high bandwidth network connections.

In order to accurately monitor network traffic, Virtual Protocol Stacks mirrors the implementation of the network protocol on the target machine. Since implementations vary by operating system, intruders can exploit these differences to route traffic past a standard intrusion detection system. Insertion/Evasion attacks used by intruders occurs when the target or intrusion detection system receives a message that differs from the one

received by the true destination. Virtual Protocol Stacks eliminates the ability for this type of attack to occur.

### *1.1.1 Host-Based Intrusion Detection Systems*

Host based approaches monitor audit logs generated by the operating system [1] and applications searching for patterns that match known malicious behavior. These attack signatures consist of a set of commands that an attacker uses to gain unauthorized access [2]. The intrusion detection system must watch for all possible permutations of a signature to ensure that all illicit activity is caught.

Another method consists of generating a standard of activity for an authorized user, such that if the activity of that user abruptly changes, it would alert the intrusion detection system to possible unauthorized access to a computer account. Since the normal user of the account has a known pattern that he/she follows, any peculiar behavior would be considered suspicious.

### *1.1.2 Network-Based Intrusion Detection Systems*

Network based intrusion detection systems observe all traffic traveling across the local network as well as examine it for suspicious activity, such as malformed packets or denial of service attacks [1]. Non-intrusive network based intrusion detection systems operate invisibly in the network with no degradation in performance, which make attacker knowledge of its location or existence nearly impossible. However, they are unable to perceive attacks that are originating on the host being attacked. The host and network based intrusion detection systems must work in concert with each other to offer maximum security.

### *1.1.3 Virtual Protocol Stacks*

Virtual Protocol Stacks are a building block to be used in constructing network-based intrusion detection systems only. They provide an accurate source of intelligence for the intrusion detection engine. Virtual Protocol Stacks are not applicable in host-based intrusion detection systems, since no network is available for Virtual Protocol Stacks to draw data from.

Intrusion detection systems operate on a packet by packet basis using network data. Unfortunately, this will allow intruders an opportunity to slip past the system by garbling the packets in such a way that only connection history would be able to detect his/her true intention [4][6].

An Insertion/Evasion attack is a method used by attackers to disguise an attack from network traffic analyzing intrusion detection systems by taking advantage of various packet handling by the network and target machine. The attacker formats network packets such that the intrusion detection system receives the transmitted message differently than the target machine, therefore causing intrusion detection systems to add or remove characters. The location and amount of characters added or removed by the intrusion detection system is dependent on the formatting of the packets and the conditions in which the system with accept or reject packets.

An example of an Insertion/Evasion attack is when a TCP packet is transmitted with an invalid TCP checksum. The target machine in return rejects the packet, but the intrusion detection system fails to compute the checksum and accepts the packet. Now, the data stream seen by the intrusion detection systems differs from the one seen by the target host.

Virtual Protocol Stacks models the network protocol implementation of target operating systems in order to prevent the Insertion/Evasion attack. By countering the inconsistencies that exist in different network stack implementations of various operating systems, Virtual Protocol Stacks do not leave the door ajar for the attacker to enter undetected.

## 1.2 Objectives

Intrusion detection systems often lack knowledge of the target machines and how they handle TCP options and packet fragmentation, causing an inconsistency between the network monitoring host and target machines. Attackers format packets such that the intrusion detection system receives one message while the target machine receives another.

By creating Virtual Protocol Stacks, intrusion detection systems will identify the inconsistencies in packet processing between the system and the target machine. By taking into account the network stack characteristics of the target machine, Virtual Protocol Stacks prevents all Insertion/Evasion attacks against the intrusion detection systems. Therefore, intrusion detection systems will have excellent situational awareness of traffic and events passing through their networks.

### *1.2.1 Packet Handling*

Packet fragmentation divides larger packets into smaller packets that can transmit over heterogeneous technologies. Due to the nature of packet switched networks, packets sometimes may arrive out of order. Therefore, for fragmentation to work properly, host machines must buffer data until a contiguous block can be passed up the Virtual Protocol

Stack. All data must be accounted for in a block before being sent to the next layer in the network stack. In order for the entire system to work, the destination machines must be able to reassemble the message in the correct order [4].

Various methods exist in the handling of packet order and packet fragmentation by the network stack. Linux will overwrite previously received data that has not been passed to the next level of the protocol stack, while Microsoft Windows NT 4.0 always keeps old data. As a result, one implementation may receive packets that were eliminated from another, causing a network based intrusion detection system host to see traffic differently then the target host [4].

To prevent incorrectly processed messages, Virtual Protocol Stacks represent different hosts' network stacks. When data packets arrive at the network connector, they are passed to the Virtual Protocol Stack that represents the destination host. The raw data packet is handled in exactly the same method, assuring that the intrusion detection system running at the application layer has a consistent view of the network [4].

### *1.2.2 Virtual Protocol Stacks*

To combat network stack implementation inconsistencies, a new software application was developed and tested, using Virtual Protocol Stacks to determine how target machines handle anomalous packets through black box testing. In this application, a set of test packets was sent to the target host, which in return verified their delivery.

After the test is complete, the sender and receiver of the test packets compared packets that made it through the stack and those that did not. From this information, a Virtual Protocol Stack was created on the intrusion detection system to represent that particular host on the

network, therefore preventing packets from slipping past the intrusion detection system by processing TCP/IP packets in the same manner as the target machines.

As the most common and reliable protocol with source code freely available, TCP/IP was used in Virtual Protocol Stacks. Since the TCP protocol is complex with various options and features, attackers can often take advantage. Virtual Protocol Stacks was created to eliminate various attacks, resulting from the complexities of implementations due to differing interpretations of the TCP specification.

By creating Virtual Protocol Stacks, intrusion detection systems will be able to identify all attacks taking advantage of inconsistencies that exist in packet processing between the intrusion detection systems and the target machine. By taking into account the characteristics of the target machine's network stack implementation, Virtual Protocol Stacks prevents misconfiguration of the network data source due to operating system differences. The personnel responsible for configuring and managing the intrusion detection systems will not have to worry about the particularities of the systems being monitored, Virtual Protocol Stacks will compensate for them. Therefore, intrusion detection systems are guaranteed accurate data on which to perform their intrusion detection algorithms.

### 1.3 Organization

By countering inconsistencies in network stack implementations and operating systems, Section 1 will discuss the Virtual Protocol Stack's ability to prevent undetected attacker entry. Virtual Protocol Stack design and implementation is reviewed in Section 2. Various benefits listed in Section 3 will prove that Virtual Protocol Stacks adapt themselves to packet handling features in operating systems and assist monitors in tracking host activity accurately.

Scenarios, such as Teardrop attacks, partial TCP connections, desynchronizing with sequence numbers, artificial time-to-live counts, and malformed TCP header options will be applied in Section 5. Virtual Protocol Stacks will then be tested for fragmented packet reassembly, and various TCP header options using the Ptacek & Newsham procedure in Section 4. Section 6 highlights related work in the area of network based intrusion detection. Finally, Section 7 summarizes the Virtual Protocol Stack's importance as a building block for network based intrusion detection systems in heterogeneous computer environments.

*Chapter 2*

# THEORY

## 2.1 Virtual Protocol Stack

Virtual Protocol Stacks are a building block for network-based intrusion detection systems. The major design goal is to provide a consistent view of data traffic passing between the attacker and the target, including IP fragmentation and TCP sequence number overlap.

To meet this goal, Virtual Protocol Stacks must account for inconsistencies in the implementations of the TCP and IP protocol in each different target system. To account for the differences, Virtual Protocol Stacks construct a representation of each host being monitored, by performing black box testing of the target. When a Virtual Protocol Stack identifies a host for which no representation exists, it executes a test procedure to determine the peculiarities of the host.

Virtual Protocol Stacks are more concerned about what data is flowing into a target machine versus what is flowing out. An attacker can generate packets that appear as if they are coming from the target machine, which could lead to a squandering of monitoring resources such a memory on the intrusion detection system. Therefore, Virtual Protocol Stacks do not monitor data transmitted by the target; it only processes inbound traffic. Once a TCP segment is reassembled with a TCP header, the destination port number is used by the IP Processor to route the segment to the appropriate TCP Processor.

Often, attackers are attempting to exploit defects or security loopholes in known computer services such as Sendmail and RPC. Since these services run with unencrypted interfaces to clients, pattern-matching intrusion detection systems using Virtual Protocol Stacks as their network data source should operate well against these sorts of attacks. Attackers would transmit their attacks in plain text, but will attempt to disguise their activity from intrusion detection systems by utilizing IP fragmentation and/or TCP segment rewrites. Virtual Protocol Stacks have the ability to reassemble TCP streams and IP fragments and unearth these onslaughts.

The Virtual Protocol Stack's architecture is broken up into five major components as illustrated by Figure 1. Each component is charged with one specific task to simplify implementation.

The Network Driver is the source of data for Virtual Protocol Stacks by sniffing packets from the local network. These packets are passed to the Packet Router, which looks for an IP Processor representing the destination system. If an IP Processor does exist, the packet is passed to it. The IP Processor examines the IP header and performs a series of checks to ensure that the packet will be accepted by the target system.

In the event the packet passes all IP Processor tests, it will then be transferred to the TCP Processor, which maps to the destination port of the target. Again a series of checks are performed, before the TCP Processor is able to attempt reassembly to the TCP data stream. Once the data stream is reassembled, the data is written to a log file specific to the destination port.

If no IP Processor exists for that address, the Tester is informed of that system and attempts connection to the Mobile Code Platform. The Client Data Sink bytecode is then

transferred to the Mobile Code Platform and executed. The Tester connects to the Client Data Sink and performs a series of tests. When the tests are complete, the data returned by the Client Data Sink is used to construct TCP and IP Processor, which are added to the Packet Router's list of Processors.

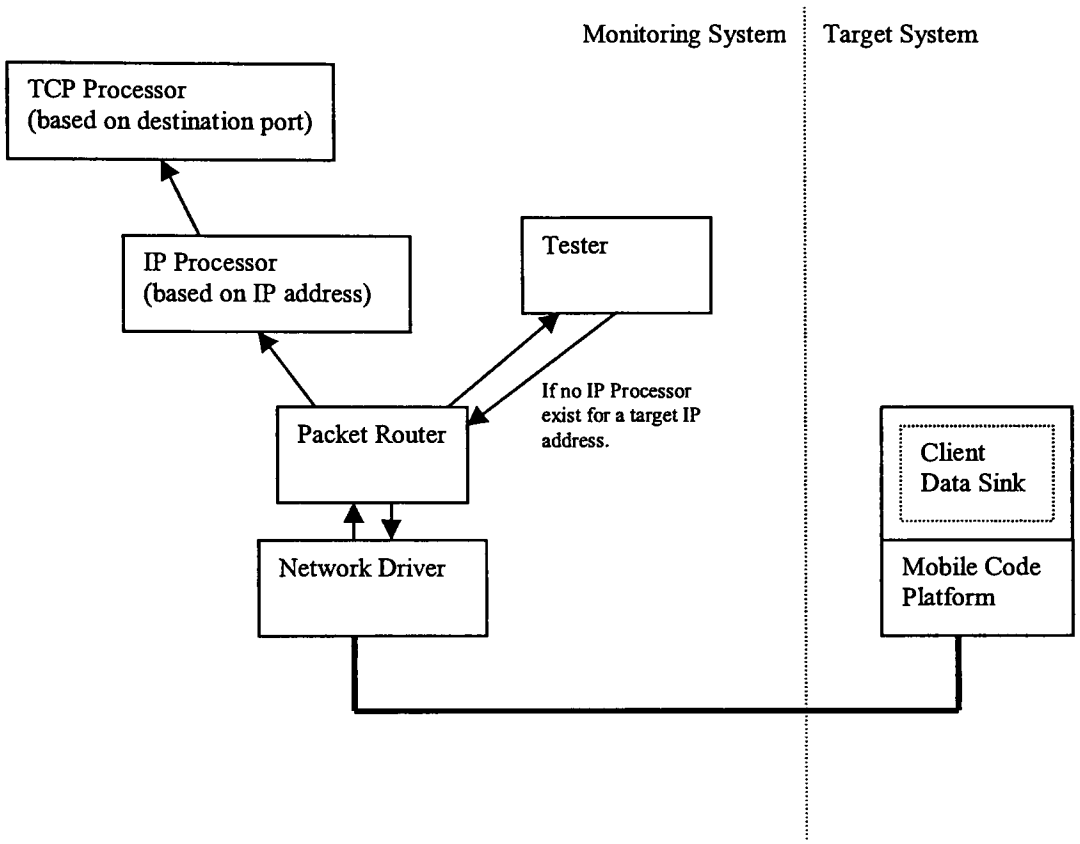


**Figure 1. Virtual Protocol Stack system architecture.**

*2.1.1 Network Driver*

The Network Driver is the system's interface to the physical network. The primary task is to promiscuously extract packets off the local network and pass them to the packet router. However, the Network Driver also provides the capability to send pre-formatted IP packets

over the network. This feature is needed by the Virtual Protocol Stack's Tester component to perform testing of target systems for different scenarios, such as IP fragmentation and abnormal TCP option flags.

The Network Driver is broken up into two pieces, containing the packet sniffer and packet injector. The packet sniffer seizes link layer packets off the network, parses off the link layer headers and sends it to the packet router. To perform this task, the packet sniffer is built using the libpcap libraries [14], which promiscuously retrieves packets off the network. Since the prototype version of Virtual Protocol Stacks is only interested in IP packets, the packet sniffer filters out non-IP packets by checking the protocol type field in the link layer header.

To inject specially formatted packets out to the network, the packet injector is used by the Tester. The packet injector is constructed using Libnet libraries [15], which allows an application to write directly to the network device. In order to have direct access to the device and all the TCP/IP header field options, the packet injector must also format the link layer header. To format the link layer header, the Ethernet addresses of the local machine and the destination machine are required. Fortunately, the packet sniffer sees the Ethernet address with every IP packet. Therefore, the Ethernet address of the destination machine is passed up with the IP packets from the packet sniffer to enable the packet injector to address packets.

To simplify the interface between the Packet Router and Tester written in Java, and the Network Driver written in C, the C components communicate with the Java components using sockets. Since the sockets are connecting local items, it generates no extra traffic for the packet sniffer. This method of interfacing the components keeps them very loosely coupled. The packet sniffer and packet injector are separate processes, which permits them to take advantage of a multiprocessor system.

*2.1.2 Packet Router*

Once the Network Driver receives a packet, it is passed to the Packet Router, which will determine the IP processor to handle the newly received packet. The Packet Router extracts the destination IP address from the IP header field (Figure 2) and determines IP Processor existence for this address. If an IP Processor exists for this address, the packet is passed to the appropriate IP Processor.

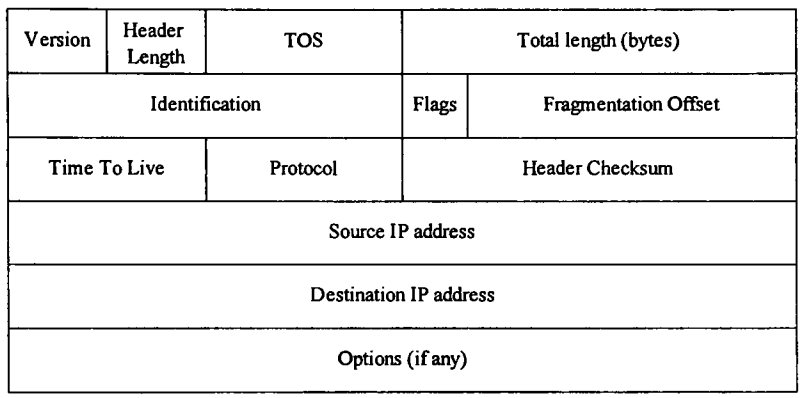


**Figure 2. Version 4 IP header [3].**

The IP header contains a variety of information about the packet sent over the network, such as source and destination address. Since the Packet Router is only concerned with passing the packet to the appropriate IP Processor, it does not look at the other fields. The IP Processor must examine and validate the IP header.

If an IP Processor does not exist for the destination address and the destination address is one that should be monitored by Virtual Protocol Stacks, the Packet Router informs the Tester that a target needs to be evaluated. Once the Tester completes the evaluation, an IP and TCP processor is constructed to represent the target and is added to the Packet Router's list of IP Processors.

Using the Packet Router to send packets to the suitable IP Processor enables the TCP and IP processor to be multithreaded. Therefore, improvements can be made in multiprocessor system performance and in the simplicity of a system architecture design. When a packet is received, the Packet Router dispatches it to the corresponding IP Processor and then prepares the next packet to be routed to a different IP processor.

### *2.1.3 IP Processor*

The principal task of the IP Processor is to manage the Internet Protocol identically to the target machine. By examining all the fields of Figure 2 such as IP version, protocol type, and fragmentation offset, the IP Processor can determine how the packet is handled.

#### *2.1.3.1 Version*

The Version field indicates which version of the IP protocol is being used in the packet. This will influence the location and method in which the IP header and data is interpreted. Currently, Virtual Protocol Stacks supports IP version 4 (IPv4). The Testers verify how the target system handles packets with version numbers other than version 4. The target machine should only handle version 4 IP headers. However, if it accepts and processes version numbers other than 4, the IP Processor must handle it.

#### *2.1.3.2 Protocol Type*

The Protocol Type field marks the type of higher level protocol that has been encapsulated by IP. The prototype version of Virtual Protocol Stacks can only handle the processing of TCP packets, and all other protocol types were discarded. Some other possible protocol types include UDP and ICMP.

*2.1.3.3 Header Length*

Header length indicates the length of the IP header in 32-bit words [3]. This value can range from 5 to 15, which translates into a size range of 20 to 60 bytes. Standard IP headers are twenty bytes in length and should never be shorter. Since the minimum size of an IP packet is twenty bytes, the Tester component of Virtual Protocol Stacks must verify activity on the target machine when the header size indicates less than twenty bytes. The truncated header length should cause the packet to be dropped, if the entire IP header is not intact or the IP checksum has failed to detect data corruption.

*2.1.3.4 Type Of Service*

The Type of Service (TOS) field is used to recommend a handling preference for the IP packet. Most IP implementations ignore this field, but newer versions are starting to make decision influenced by these bits [3]. Currently four of the bits are used to signify minimum delay, maximum throughput, maximum reliability, or minimum monetary cost.

The default standard service is represented when no handling preferences are selected, while one selected flag corresponds to the previously mentioned options. Only one flag should be enabled at all times. Since this field effects routing more than data processing, the IP processor does not perform any validation of this field.

*2.1.3.5 Total Length*

The Total Length field denotes the total length of the IP packet in bytes [3]. The IEEE 802.3 Ethernet frame format has its own length field [10] and the checksum used by IP only covers the IP header. The Tester component of Virtual Protocol Stacks must test a target machine for how it determines the total length of the IP packet. If the IP checksum covered the data, the checksum would fail if it misunderstood the transmitted packet size. An intrusion detection system that monitors data at the TCP layer could reject data if the IP

layer added extra data or removed relevant data. In either case, the TCP checksum would fail and the data would be discarded, which is an example of an Evasion strategy. The packet length could either be based on the Total Length field or the total size of the IP packet sent up from the Ethernet data link layer minus the size of the IP header.

#### *2.1.3.6 Identification*

The Identification field is used to uniquely classify each packet sent out on the network by a host [3]. Both the identification field and source IP address guarantee that no other system on the network will generate a packet with identical values. The value for this field is normally generated by the IP implementation of the sending host [3].

The operating system keeps track of a system variable used as the identification and each time a packet is transmitted, the variable is incremented [3]. Virtual Protocol Stacks need only handle this field when IP fragmentation has occurred. When a TCP segment is fragmented into multiple IP packets, the Identification field is copied into each IP header that belongs to the fragmented TCP segment.

#### *2.1.3.7 Flags*

One flag announces that the IP packet should not be fragmented, this flag is referred to as the "Don't Fragment" flag or DF. If the IP packet is larger than the maximum size allowed by the network medium, the IP packet is dropped if this flag is set [3]. The other important flag used declares that the IP packet is part of a larger fragmented packet. All packets that belong to a larger fragmented packet have this flag set, except for the last packet in the series [3]. Virtual Protocol Stacks need to evaluate these flags when handling fragmentation.

*2.1.3.8 Fragmentation Offset*

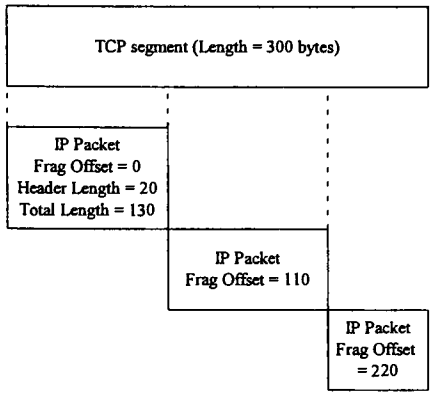


**Figure 3. Fragmentation offset example. Header length applies to all IP packets.**

To reassemble fragmented packets, the IP protocol uses the Fragmentation Offset field to determine where this fragment fits in reference to the beginning of the original unfragmented packet [3]. A packet is fragmented when its size exceeds the maximum transfer unit (MTU) of the link layer protocol, such as Ethernet, which has a maximum transfer unit of 1500 bytes [11].

In Figure 3, the example TCP segment is 300 bytes in length, while the maximum transfer unit for the underlying network is 130 bytes. To send this TCP segment, it must be fragmented into three IP packets. Each IP packet, except for the last, contains 110 bytes worth of the TCP segment. Note that the initial IP packet has a fragmentation offset of 0, this corresponds to the first bytes of the TCP segment. The second IP packet has a fragmentation offset of 110, which coincides with the byte location of the first byte of the second IP packet's data in the TCP segment.

An attacker could use this field to execute an Insertion/Evasion strategy against a network, by forcing overlap or rewrite of the fragmented packets. To do this, the attacker would need to be aware of the operating system the target host is running in order to understand the precedence for data when overlap occurs. Virtual Protocol Stacks will need to duplicate the way in which the target machine deals with IP fragment rewrites.

#### *2.1.3.9 Time To Live*

The Time to Live (TTL) field is used by the network to limit the amount of time that a packet can bounce around the network. The value in this field determines how many routers a packet can pass through before it is dropped [3]. Since Virtual Protocol Stacks are designed to work on a local network, with no routers between it and the target machines, this field is ignored.

#### *2.1.3.10 Checksum*

The Checksum field is used to detect errors in the IP header. To calculate the checksum, one's complement addition is used to sum the contents of the header [3]. This checksum is inserted into the IP header before transmission. On the receiver side, the host sums the header in the same fashion but includes the value in the Checksum field. If there are no errors in the header, the resulting sum should be binary ones [3][16].

When an error does occur, the packet is discarded. Bad checksums occur since the media that the packets traverse are not free of errors, especially when the packets pass over wireless connections. IP assumes that the higher level protocol is performing its own error detection on the data. Virtual Protocol Stacks also verifies the IP checksum.

*2.1.3.11 Source and Destination Addresses*

Finally, the IP header contains the source and destination IP addresses. The Packet Router uses the destination address to route the data to the correct IP Processor. Since the source address and/or port number can easily be falsified, Virtual Protocol Stacks can not determine if it is valid or not. The attacker cannot falsify the destination address since that is the only reason that the packet is routed to the target machine.

*2.1.4 TCP Processor*

The TCP Processor reassembles the application data stream following the TCP standard and maintains state of the connection according to the TCP server side state diagram in Figure 6. The states of the diagram refer to the state of the server side socket. In the client/server paradigm, the server listens on a socket for connections and responds to requests from the client. A socket is the name given to the mechanism in which the client and server pass data to each other. There must always be a socket pair for communications to occur.

In addition to monitoring connection state, the TCP Processor examines a variety of fields in the TCP header. These fields help determine the state of the TCP connection, indicate applications on the destination system that data is routed for, and recognize clues necessary to determine an order to the data. The effects of invalid TCP header fields are undefined by the TCP specification and vary by implementation. Virtual Protocol Stacks bring definition to these undefined scenarios and provide data consistency for an intrusion detection system.

*2.1.4.1 TCP Header*

The main purpose of the TCP header is to ensure that packets are routed to the proper application and are delivered in order reliably. The port number fields of the TCP header are used to signify which application is the recipient of the data and sequence numbers are used to label each byte sent over a connection.

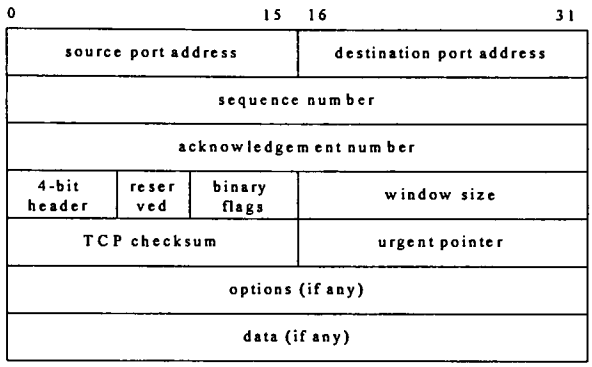


**Figure 4. TCP header [3].**

To ensure reliable data transmission, TCP uses sequence numbers to enumerate each byte sent over a connection. The sender transmits a number in the sequence number field that corresponds to the first byte contained in the packet. When the data is received on the other end, it acknowledges with a sequence number that coincides with the next expected byte. That acknowledgement number is placed into the acknowledgement field of the TCP header.

The sequence numbers used for a connection do not necessarily start at zero to prevent repeat packets from floating around the network. The initial sequence numbers are generated by the network drivers and are agreed upon by the sender and receiver at the start of a connection. For additional reliability, the TCP header contains a checksum of the entire packet to ensure no data corruption has occurred.

The TCP header's window size field allows the receiver to inform transmitters of the maximum number of bytes willing to be accepted, according to size [3]. This value is constantly updated while the connection progresses, since the receiver is capable of buffering data. Virtual Protocol Stacks tests the target host with various combinations of TCP header options to discover dropped data, and which assures that the network based intrusion detection system was never blinded.

*2.1.4.2 TCP Binary Option Flags*

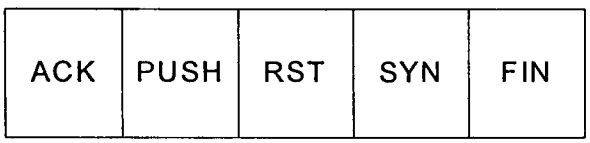


**Figure 5. TCP Binary Option Flags extracted from TCP header [3].**

Binary flags, on the other hand, are only valid at certain times, meaning intrusion detection network stacks must be aware of dropped data due to invalid flags. Currently, six flags exist, with more space reserved for future extensions. The first flag from the Bit 0 edge is the urgent flag or URG, which is used to inform the receiver that urgent data exists in this packet. The receiver must then determine how to handle urgent data [3]. Urgent pointer fields are used as an offset from the sequence number of the received packet to indicate the last byte of urgent data [3]. Virtual Protocol Stacks currently do not perform any processing dependent on the URG flag.

*2.1.4.3 ACK*

The next flag is acknowledgment or ACK flag, which is used to indicate that the value in the acknowledgment number field is valid. It should coincide with the sequence number of the next valid byte of data that the receiver is expecting and implies that all bytes prior to this sequence number have been successfully received. The TCP Processor of Virtual

Protocol Stacks uses this flag in conjunction with the SYN flag to determine the establishment of connections.

*2.1.4.4 PUSH*

Following the ACK is the push flag or PUSH, which is used by the transmitter to inform the receiver to push the data through the buffers as quickly as possible. Berkeley-derived implementations of TCP normally ignore the push flag, because data is normally pushed through as soon as possible [3]. PUSH is ignored by the current version of Virtual Protocol Stacks.

*2.1.4.5 RST*

After the PUSH flag, the reset or RST flag is executed, which is employed to immediately terminate a TCP connection. Normally, reset packets are generated when an invalid packet is received by a connection [3]. Under these circumstances, a port could receive a SYN+ACK (see Section 1.2.3.4) packet for a connection that was never initiated. The receiver will then send a RST packet to inform the sender that the connection is in error and should be terminated. The TCP Processor watches for RST packets as a method for tearing down an established connection. This flag will set the TCP Processor state machine to the LISTENING state.

*2.1.4.6 SYN*

Next to the RST flag is the synchronization or SYN flag that is used to establish a connection during the three-way handshake. When a client wishes to make a connection to a remote host, he/she sends a SYN packet with an initial sequence number. The server must acknowledge the sequence number with a SYN+ACK packet with both the client's

sequence number and the server's sequence number. To finalize the connection, the client transmits an acknowledgement of the server's sequence number and the connection is complete. For Virtual Protocol Stacks to believe that a connection has been established between two systems, Virtual Protocol Stacks must see the initial SYN packet followed by the SYN+ACK packet sent as a response. These packets provide Virtual Protocol Stacks with a starting sequence number in which to reassemble the TCP data stream.

The SYNC+ACK packets are also used by the Tester component to acquire the initial sequence number for the test connection. Before the Tester can start sending test packets to the Client Data Sink, it must discover the sequence number for the connection between the two components. The Tester performs this task by establishing a standard stream connection to the Client Data Sink and uses packet sniffers to hear the SYN+ACK packet and obtain the connection sequence numbers.

*2.1.4.7 FIN*

Finally comes the finish or FIN flag, which is used to gracefully terminate a connection. To close a connection both the client and the server must send FIN packets and receive the acknowledgments for them [3]. The TCP Processor state machine determines when a connection is terminating uses the FIN flag.

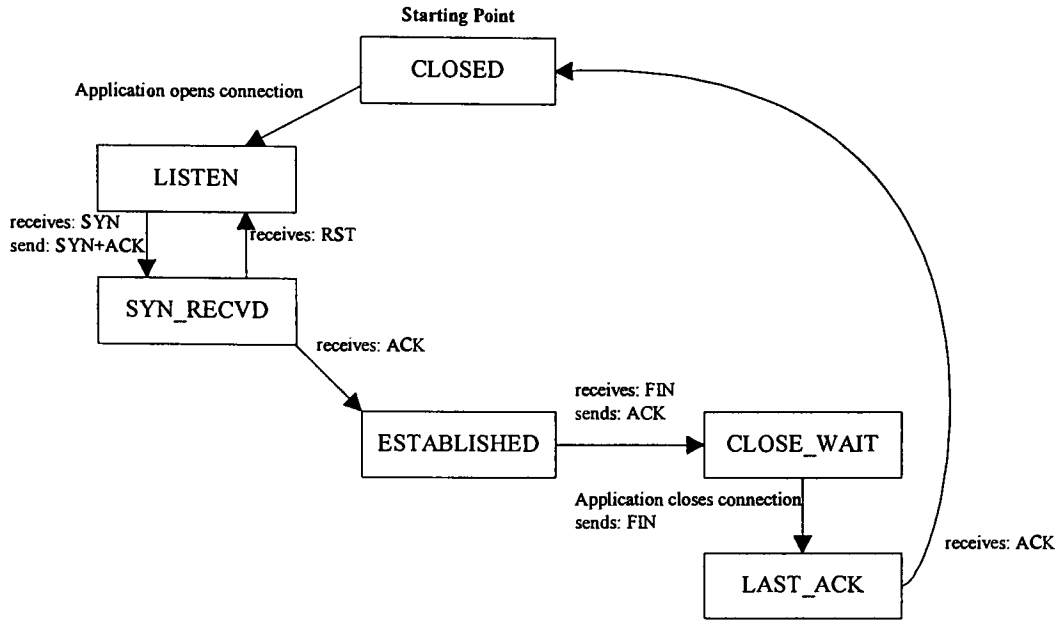


**Figure 6. TCP connection state diagram [3].**

All sockets on a computer system start in the CLOSED state, where no communications can occur. To enable application service, the application must bind itself to a port and listen for a client to establish a connection, this moves the socket state to LISTEN. Here, a client can attempt to connect to the server by sending a TCP packet with the SYN flag set. The server responds using a packet with both the SYN and ACK flag set.

Once the SYN is received, the socket moves to the SYN_RECVD state, waiting for either a packet with the RST or ACK flag set. If the RST packet is received, the socket backpedals to the LISTEN state. If an ACK is received, the socket state moves to ESTABLISHED and the client and server are capable of passing data. This connection establishment procedure is known as the TCP three-way handshake.

To terminate a connection, the client sends a TCP packet with the FIN flag set. On reception of the FIN packet, the server side sends an ACK packet, the application closes

its socket and then sends a FIN packet. The FIN packet sent by the client moves the socket state to CLOSE_WAIT. Once the application closes the socket and sends a FIN packet, the socket state changes to LAST_ACK. When the client acknowledges the server's FIN packet with an ACK packet, the connection is officially terminated. This style of termination is referred to as passive close [3], which is normal operation for a connection.

Another type of connection termination is the active close [3], where the connection is stopped due to the server application closing the socket. The application could close the socket due to an application error or the application is being shutdown. Instead of the client closing the connection, the server does it. Figure 7 illustrates the active close state diagram.

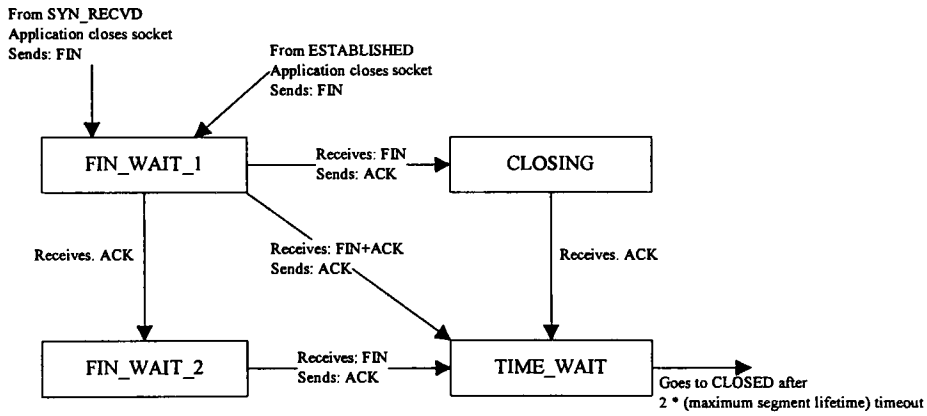


**Figure 7. TCP Active close state diagram [3].**

If for any reason the server application decides to close the socket, a FIN packet is generated and sent to the client. The condition changes the socket state to FIN_WAIT_1. From here, the client can react in a variety of ways to terminate the connection. The client may acknowledge the FIN and then send its own FIN packet. It may also combine both ACK and FIN into one packet and terminate the connection.

Any of the paths will eventually get the server to the TIME_WAIT state, where the server waits two times the maximum segment lifetime before going to the CLOSED state [3]. In most implementations, this timeout ranges from thirty seconds to two minutes [3].

The maximum segment lifetime is the maximum amount of time an IP packet can keep bouncing around the network before it is dropped [3]. This prevents stale packets being received on the same port for a different instance of a connection. The maximum amount of time a packet can stay alive is bounded by the Time to Live field, which determines how many routers the packet can pass through before it is dropped.

The TCP Processor of Virtual Protocol Stacks must keep track of the connection states to accurately process data coming into Virtual Protocol Stacks. A future direction for Virtual Protocol Stacks is to test for data passing during TCP states other then ESTABLISHED.

*2.1.5 Tester*

The Tester component evaluates a target machine's TCP/IP implementation, as well as constructs the TCP and IP Processors used by Virtual Protocol Stacks. Once the Packet Router has determined that no IP Processor exists for a target machine, it informs the Tester. The Tester then attempts to connect to the target machine's Java mobile code platform and send it the Client Data Sink application.

The Client Data Sink application is a small Java class, which is transported to the host under test. It is used as an endpoint for test packets sent by the Tester. An assumption made by Virtual Protocol Stacks is that there is a mobile code platform running on the target host, which accepts the Client Data Sink application class. When the Client Data Sink code is resident on the target, it starts up a command and control connection server, which

the Tester uses to establish connection to the Client Data Sink. This enables the Tester to query the Client data Sink for results and status.

Next, the Tester starts executing each test scenario from its library. These scenarios determine how the target machine's TCP/IP implementation reacts to certain conditions. Before each scenario, the Tester informs the Client Data Sink application that a test cycle is beginning. Once notified, the Client Data Sink starts up a test server that is connected to by the Tester. This test server is where the evaluation packets are sent and buffered by the Client Data Sink. The test scenario is complete when the Tester terminates the test connection. After each scenario is executed, the Tester retrieves the results from the Client Data Sink via the command and control connection. The results are the test string that the Client Data Sink received during the test cycle. Once all of the scenarios are complete, the Tester constructs a TCP and IP Processor representation of the target machine based on the results of the tests. The IP Processor encapsulates the TCP Processor. The newly created IP Processor is appended to the Packet Router's list of IP Processors and the target machine is ready to be monitored.

The test library contains a variety of scenarios for evaluating the target host, covering IP fragmentation, invalid IP header options, TCP sequencing, and invalid TCP header options. During the test, the Tester attempted to send a known character string to the target using the conditions prescribed by the scenario. Based on the message received by the target, the Tester constructed an accurate representation of the target's network protocol implementation.

For testing during development, a Linux system running RedHat beta release 7.0.90 (Fisher) and a Windows Millennium Edition system were used. These two systems were chosen for their availability and reported differences with respect to TCP segment and IP fragmentation reassembly.

*2.1.5.1 IP Fragmentation*

The IP Fragmentation test breaks the test character string into 8 byte IP fragments and sends them to the target. This test determined if the target machine possessed the capability to reassemble fragmented IP packets, because if it does not, the test message would have been lost. The IP protocol standard indicates that all hosts shall be capable of receiving and reassembling fragmented packets [13]. This test verified that the host does indeed handle fragmented packet so that Virtual Protocol Stacks does also.

*2.1.5.2 IP Fragment Rewrite*

Once it is determined that the host handles fragmented packets, the Tester attempted to overwrite previously received data, by sending two fragments with identical fragmentation offsets. Note, for fragmentation rewrite to work correctly, the attacker must also ensure that the TCP checksum is modified so that if the new data is accepted, the TCP layer will not reject the packet due to TCP checksum failure. Rewrite occurs when the data from a previously received packet is overwritten by data from a new packet. Depending upon the IP reassembly algorithm, previously received data might be overwritten with new data or the new data is dropped.

An attacker could use questionable handling of rewritten data to his/her advantage to change the content of messages before they are handed to the application layer. If the intrusion detection system replaces old data with new data while the target favors old data, there is a loss of reality by the intrusion detection system.

If IP fragmentation rewrites are not handled properly, an intrusion detection system could be easily misled. The attacker could be downloading the password files, while the intrusion detection system thinks the attacker is downloading a file called "homework". Running the

IP fragmentation rewrite test allowed Virtual Protocol Stacks to determine how the target handles fragment rewrite. From the results of this scenario, the Tester constructed an IP processor that handles fragmentation identically as the target.

#### *2.1.5.3 Invalid IP Version*

In order to process IP headers correctly, an IP implementation should be confirm the version field of the IP header. The location and size of the header field could change from version to version of IP. Virtual Protocol Stacks is currently implementing version 4 of the IP protocol so the targets that it is monitoring should also be running version 4. To verify that the target ignores packets with invalid version identities, the Tester sent a test packet with an invalid version identifier. If the target does not validate the version, Virtual Protocol Stacks will mirror this lapse. For the Tester scenario, an IP version of three was sent to both evaluation systems.

#### *2.1.5.4 Invalid IP Header Size*

In the IP header, a field is used to inform the receiver of the length of the header, since there are optional fields in the header. The size of the header is described in terms of 32-bit words [3]. At minimum, five 32-bit words are required for a deliverable packet. For this test, Virtual Protocol Stacks attempted to determine whether the packet was discarded correctly or the message was accepted when the size in the header field is less than five words. Based on the results of the test, Virtual Protocol Stacks verified the header size in accordance with the target.

#### *2.1.5.5 IP Packet Data Length*

Two methods exist in which the target machine's IP implementation can determine the amount of data in an IP packet. The first method involves subtracting the IP header length from the value contained in the IP header's total length field. The second method uses the IP packet length passed up by the data link layer, followed by the subtraction of IP header length. The latter method violates the layered design of the protocol stack, in which the network stack designed, but is still a plausible method for calculating the length of the data portion. Virtual Protocol Stacks sent a message using a total length that is shorter than the actual message, which determined how the amount of data is calculated and how it was done by Virtual Protocol Stacks.

#### *2.1.5.6 Invalid IP Checksum*

When messages are transmitted over a medium that is capable of errors, a method must exist for detecting these errors. If no error detection is done, the computer could behave unexpectedly due to faulty input. The IP header contains a checksum field that is used to validate the header contents when it is received by the target. In Ptacek and Newsham's tests, a few of the tested intrusion detection systems, such as Internet Security Systems' RealSecure version 1.0.97.224, did not validate the IP checksum [4]. For completeness, Virtual Protocol Stacks verified that the target host validates checksums so that no assumptions are made about the target.

#### *2.1.5.7 TCP Sequence Numbers*

To send a reliable message, TCP uses sequence numbers to keep track of each byte sent over the link. From the sequence numbers, TCP can determine if any bytes have been lost and need to be retransmitted. As a baseline for evaluation, the Tester component of Virtual Protocol Stacks transmitted a character string using 1-byte TCP segments.

*2.1.5.8 TCP Segment Rewrites*

Similar to IP fragmentation overlap, this test attempts to replace previously received data by repeating a TCP segment with different data. The first instance of the segment contained data that is different then the second instance of the segment. This may cause the message to appear differently depending upon the method used by the target host to determine which data to use. One implementation may keep the first copy of the segment, while the other may overwrite the old segment with data from the new segment. From this test, Virtual Protocol Stacks assigned priority to either new or old data dependent on the result.

*2.1.5.9 TCP Segment Overlap*

A TCP segment is made to overlap a portion of previously received data. This test is identical to IP fragmentation, except that it is done using the TCP sequence numbers. The Tester component sent a known character string using multiple TCP segments, which were not sent in order to ensure that the TCP segment about to be overwritten has not been processed and passed up to application layer.

When the target's segment has been transmitted, another segment was sent that belongs before the target's segment in the reconstructed data stream. This newly sent packet overlapped a portion of the target segment's data. If the target host favored old data, the string returned by the Client Data Sink application would be the original string, else it would have been the modified string. Based on the string returned by the Client Data Sink application, Virtual Protocol Stacks determined if the TCP Processor must favor old or new data.

#### *2.1.5.10 Invalid TCP Checksum*

In the TCP header, a checksum field exists that is used for error detection of the TCP segment. Unlike IP, the TCP checksum protects the header and data. The target machine should be verifying that the checksum matches the received data. Virtual Protocol Stacks evaluated whether or not the target host calculates and verifies the TCP checksum.

#### *2.1.5.11 Invalid TCP Header Length*

The TCP header contains a field to specify the header length, which allows TCP to have a variable length header. A minimum of 20 bytes are needed for the TCP header for correct operation. The length is specified in the header by number of 32-bit words, therefore the minimum number in the header should be 5. The Virtual Protocol Stacks tested the target reaction to TCP segments with invalid header sizes.

### 2.2 Client Data Sink Application

To perform black box testing of a target machine's TCP/IP protocol implementation, Virtual Protocol Stacks needs a simple application with a small memory footprint to receive data while executing the tests. For Virtual Protocol Stacks, there are two distinct pieces, a mobile code platform for executing Java and the client data sink. The mobile code platform is a service application that runs on the system being monitored, which provides a sanctuary for the client data sink to download and execute.

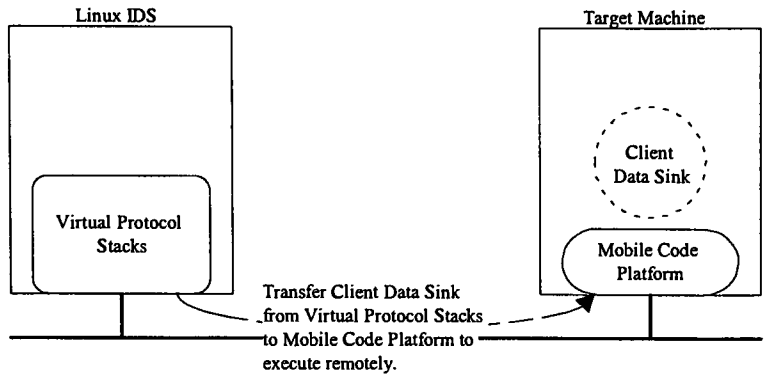


**Figure 8. Example of client data sink movement to target host using mobile code platform.**

### *2.2.1 Mobile Code Platform*

The mobile code platform is a network server application that enables applications to be received by the remote host and executed. The target machine runs the mobile code platform service so that the Tester component of Virtual Protocol Stacks can transfer the client data sink application to the target. The mobile code platform makes Virtual Protocol Stacks easier to upgrade and provides limited security by not permanently installing any Virtual Protocol Stack applications on the target machine.

Since Virtual Protocol Stack applications are not permanently resident on the remote host, host upgrades are never required. Virtual Protocol Stacks transfer the remote client data sink application to the mobile platform, so that updates done to the client data sink application are limited to the Virtual Protocol Stack application itself. Therefore, the mobile code platform should never need upgrades other than security enhancements.

Using mobile code provides a limited form of security, since the executables used by Virtual Protocol Stacks for testing are not resident on the remote machine until downloaded. The executables are only downloaded when a Virtual Protocol Stack test is required and is removed once the tests are complete. The attacker will have little chance to corrupt the executables since the client data sink is not permanently resident on the target machine. Virtual Protocol Stacks download the client data sink for every test run, which guarantees the code is not contaminated.

Java was used to limit the security risk of mobile code. Java provides runtime verification before execution to ensure that the application is behaving appropriately [12]. This helps to ensure that the code loaded by the mobile platform cannot behave maliciously.

Java limits the programmers' ability to handle memory allocation, therefore providing security to a computer system [12]. Programmers cannot create pointers to memory to circumvent access control. Java performs its own garbage collection, which releases unused memory that has been allocated by an application. This helps minimize waste of resources by an application.

To make Java secure for distributed computing, Java separates classes that have been imported over the network from local trusted classes [12]. This guarantees that remote classes do not replace local trusted classes.

Java provides a secure area, called a sandbox, in which Java applications are executed [17] [18]. This sandbox enables the Java Runtime Environment to control access to the external system through the Java Application Programmer's Interface. The Java API enables the application to access the operating system services through the Java Virtual Machine in a uniform but controlled manner [18].

Java is therefore an ideal language for implementing the mobile code platform due to built in security and distributed computing capabilities.

### *2.2.2 Client Data Sink Application*

Client data sink is the mechanism used to retrieve results of a test on a target machine. The client data sink application acts as a remote sensor for Virtual Protocol Stacks, which gathers data on the remote target machine. When a target machine is being evaluated, the client data sink is needed to retrieve the test data.

Once a new target is identified, the Tester component of Virtual Protocol Stacks transfers the client data sink application to the remote mobile code platform where it is executed. The Tester establishes a command and control connection to the client data sink, and then proceeds with the evaluation of the target host's TCP/IP implementation.

To evaluate a target, the Tester attempts to pass data to the client data sink using a variety of different TCP and IP packets, such as IP fragmentation, TCP segment overlap, and invalid sequence numbers. At the conclusion of each test scenario, the Tester obtains the results from the client data sink. From these results, the Tester constructed a representation of the target's TCP/IP implementation. Constructing a representation of the target host in the intrusion detection system is significantly less intrusive on the performance of the protected network and system, since Virtual Protocol Stacks acts as a passive monitor after the tests have been performed. The client data sink is the ear of Virtual Protocol Stacks during assessment of a target host, while the mobile code platform is the haven on which the client data sink runs.

*Chapter 3*

# BENEFITS

## 3.1 Heterogeneous Environments

A heterogeneous environment consists of a collection of computer systems with different architectures and/or operating systems. These contrasting systems contain operating systems that behave differently, which in a network can lead to irregular packet handling capabilities. When packets are processed inconsistently, the intrusion detection system cannot guarantee knowledge of all activity occurring on the network. Without a reliable source of information and data, the network will not be totally secure.

By developing this software, network based intrusion detection applications will have a correct view of the traffic as seen by the hosts under guard, while providing an inexpensive solution for dynamic, heterogeneous environments. Virtual Protocol Stacks adapts themselves to packet handling features of any operating system, by constantly monitoring for new hosts and reexamining known hosts.

To maximize portability and coverage of Virtual Protocol Stacks, Java is used for the client data sink application to execute on the target machine. When running the training portion of Virtual Protocol Stacks, a remote client application needs to be present on the target system to gather data. No special drivers are needed for this application, because it simply receives data from the network and reports results back to the Virtual Protocol Stack application. For the purpose of running on multiple platforms, Java was chosen due to its platform independent features.

Figure 9 illustrates programming languages used for the Virtual Protocol Stack implementation. The packet-sniffing driver used by Virtual Protocol Stacks is written in C for speed and reuse of freely available code. The Virtual Protocol Stack that handles actual TCP and IP protocols are written in Java for simplicity and language features such as multithreading.

The shaded areas consist of target machine evaluation, while the white blocks are part of the Virtual Protocol Stack. Currently, most platforms are capable of running a Java virtual machine. Therefore, the combination of Java portability and global support provides maximum extendibility in heterogeneous environments.

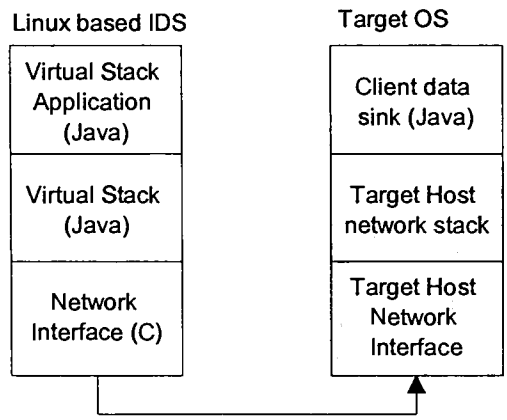


**Figure 9. Virtual Protocol Stack implementation layers.**

### 3.2 Cost

One benefit that arises from the Virtual Protocol Stack approach is cost. Traditional methods analyze the network stack source code to understand potential packet loss. Analysis must be completed on every type and version of operating system, however in many cases, the source code to the network stack implementation is not freely available.

Therefore, additional funds would be needed for source code licenses and overtime pay for extra time spent on code review. In addition, even if the time is spent to perform source code analysis, the source code may be interpreted incorrectly which defeats the purpose of the exercise.

Fortunately, Virtual Protocol Stacks do not require the analysis of code, because it tests through experimentation. Carefully designed test packets are sent to the client data sink to evaluate the target machine's network protocol implementation. Based on the results of these tests, a Virtual Protocol Stack representation was constructed to model the data handling characteristics of the target machine, and therefore eliminates the need for additional funds.

Once Virtual Protocol Stacks have evaluated a host and constructed a representation of the TCP/IP implementation, it becomes a passive network monitor. In some environments, wasted network bandwidth is undesirable. With no extraneous traffic generated by Virtual Protocol Stacks, the network load will be unaffected by its presence and be undetectably by an intruder.

### 3.3 Adaptive Configuration

Accurate network monitoring is a necessity for intrusion detection systems, because when the system is inaccurate, attackers can easily blind it. If the attacker studies the characteristics of the network protocol implementations of the target system and the packet handling of the intrusion detection system, he/she will be able to evade the system by taking advantage of the differences. This approach is commonly called Insertion/Evasion.

An intrusion detection system that uses dynamic adaptable network stacks eliminates the attacker's ability to evade the system and minimizes configuration error by the user. Since

the adaptable stacks mirrors the packet handling attributes of the machine under guard, the attacker will not be able to confuse the system using Insertion/Evasion tactics such as IP fragmentation or segment overlap.

With the expansion of computing resources coupled with the variety of operating system types and versions, system administrators have difficulty keeping track of all activity. Therefore, opportunities are open for attackers to strike. As long as the client data sink can operate on the target machine, Virtual Protocol Stacks attends to the variations of network protocol implementations and protect the network from unnecessary attacks.

When a user configures a security system incorrectly, even the strongest security system will act as if it were non-existent. Incorrectly, configured security systems will leave loopholes in the defenses that attackers can exploit. Virtual Protocol Stacks require no user configuration, because it autonomously adapts itself to the existing network implementations, therefore minimizing the possibility of configuration error.

Since the Virtual Protocol Stack application is dynamic and monitors network traffic, it detects new hosts once an attempt is made to receive traffic. Once the new host is identified, the network stack evaluation code can be dispatched for testing and a Virtual Protocol Stack representation will be created for that host.

*Chapter 4*

# TEST METHODOLOGY

Virtual Protocol Stack implementation testing examines the consistency between the monitoring host and the target host. Test points were based on the criteria used by Thomas H. Ptacek and Timothy H. Newsham in "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection" [4]. Using this experimentation procedure, testing results showed comparisons between four commercial intrusion detection products and their traffic monitoring weaknesses. The Virtual Protocol Stack software solves the shortcomings of these commercial products.

The Ptacek & Newsham procedure tested for fragmented packet reassembly, and various TCP header flags set. The goal of the tests was for the intrusion detection systems to accurately reconstruct the string "GET /cgi-bin/phf?". The validity of the test scenarios was based on whether or not the 4.4BSD TCP/IP network driver would receive the intended string [4].

For fragmentation tests, Ptacek & Newsham broke a TCP packet containing the PHF string into 1-byte fragments and sent them in various combinations, such as in-order, out-of-order and selective repeats. In an attempt to pass the PHF string before the TCP connection was fully established, such as sending the string without the final SYN+ACK packet, two of the four systems reported an attack for this invalid case. These tests also desynchronized the intrusion detection system by interleaving abnormal packets with random sequence numbers or the SYN flag set. By desynchronizing the intrusion detection system, the hope was it would stop monitoring the connection.

The major weakness of the four tested intrusion detection systems was fragment reassembly. Neither ISS's RealSecure (version 1.0.97.224 for Windows NT), WheelGroup Corporation's NetRanger (version 1.2.2), AbirNet's SessionWall3 (version 1, release 2, build 1.2.0.26 for Windows NT), nor Network Flight Recorder's NFR (version beta-2) handled fragmentation successfully. A FreeBSD 2.2 system was used as the target machine in the Windows NT environment. Each operating system has distinct methods for processing certain packets, and these differences explain some of the failures experienced by the intrusion detection systems [4].

Denmac Systems evaluated these intrusion detection systems again in 1999, where only the Network Flight Recorder's NFR was capable of handling the fragmentation attacks. The major differences between the Denmac Systems' test and Ptacek & Newsham's test were the environment and version of software. Denmac System's used the latest version of software, but only tested the systems in a Windows NT network with no account for heterogeneous environments.

Virtual Protocol Stacks was able to account for the differences in the computing systems, by adapting to all the characteristics of the individual systems. In return, by handling the differences, the fragmentation attacks will be detected and eliminated [6].

Connection monitoring must be completed to ensure consistent data handling with the target host. Some network monitoring systems start examining data that is ultimately dropped by the target host, this constitutes an insertion attack. However, if the network monitor does not examine the data until the connection is complete, an evasion attack could result, where the target machine is seeing traffic that the monitor is not. The Ptacek & Newsham tests demonstrated that Virtual Protocol Stacks eliminated insertion/evasion attacks on application level intrusion detection systems.

The evaluation environment for Virtual Protocol Stacks consisted of three computers connected together using a non-switched Ethernet hub. Figure 10 illustrates the environment for developing and testing Virtual Protocol Stacks. The attacker and Virtual Protocol Stack ran on separate Linux machines. The capabilities of Virtual Protocol Stacks were tested against a target machine with both the Linux and Windows operating systems.

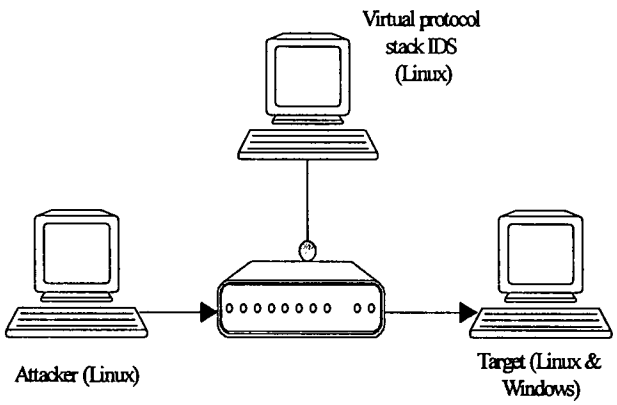


**Figure 10. Development and evaluation environment for Virtual Protocol Stacks.**

The scenarios discussed were used as a test procedure for Virtual Protocol Stacks, which allowed for a more direct comparison with the results found by Ptacek & Newsham and Denmac Systems. Each scenario generated by Ptacek & Newsham are valid in terms of real-world probability of occurrence. This test will highlight the benefits of Virtual Protocol Stacks, such as fragment reassembly and TCP header options that are processed identically with the target.

*Chapter 5*

# VIRTUAL PROTOCOL STACK EVALUATION

The evaluation environment used consists of a local area network (LAN) connected together over 10Mbps Ethernet in a star configuration using a 4-port hub. Figure 10 highlights the development system. This hub repeats all incoming traffic on each outgoing port, enabling one connected computer to simultaneously run the Virtual Protocol Stack application and act as the intrusion detection system. Each scenario run against a host on the network, verifying that the Virtual Protocol Stacks receive the same traffic as the host at the application layer.

Implementing Virtual Protocol Stacks in Java allows for maximum portability with built-in support for multithreading and distributed computing. Multithreading provides higher performance when handling multiple packet streams through the IP and TCP Processors by separating the execution of each instance.

Distributed computing enhances the Tester component's capability to evaluate the target machine through support for mobile code. When the Tester wishes to assess a host, it first must download the Client Data Sink application to the host in order to gather data. With Java's built-in support for multithreading and distributed computing, it provides an efficient and flexible solution for implementing Virtual Protocol Stacks.

Network drivers that send and receive packets promiscuously are implemented in the C programming language. When a network interface adapter is behaving promiscuously, it listens to all packets that travel along the local network.

C was chosen to allow for reuse of public domain software, such as libpcab [14] for operating the network interface adapter promiscuously. Much work has already been completed in this area, which need not be repeated. Another open software library is called Libnet [15] provides the ability send customized network packets, which is needed to review the target host under certain conditions such as IP fragmentation and abnormal TCP header options.

To evaluate Virtual Protocol Stacks, derivatives of Ptacek and Newsham's tests were used. Table 1 summarizes the results of each test executed. The tests provide a spectrum of approaches that are known to confound network based intrusion detection systems, such as fragmentation tests, partial TCP connections, desynchronization tests, and invalid header options. The scenarios used to evaluate Virtual Protocol Stacks are similar to the tests executed by the Tester component of Virtual Protocol Stacks, whose results are used to construct the TCP and IP Processors that represent the target machine. The goal was to demonstrate that Virtual Protocol Stacks adapted to the distinct characteristics of the different target systems under test.

| Section | Test Description | Test String (1st mesg/2nd mesg) | Linux | Windows |
|---|---|---|---|---|
| 5.1.2.1 | 8-byte IP fragments in order | hell,o | hello | hello |
| 5.1.2.2 | 8-byte IP fragments out of order | hell,o | hello | hello |
| 5.1.2.3 | 8-byte IP fragments with rewrite | hell,o/xxxx- | xxxxo | hello |
| 5.2.2.1 | 1-byte TCP segment with single repeat | h,e,l,l,o | hello | hello |
| 5.2.2.2 | 1-byte TCP segment with rewrite | h,e,l,l,o/--m- | hello | hello |
| 5.2.2.3 | 1-byte TCP segments with overlap rewrite | h,e,l,l,o/-xx- | hexxo | hello |
| 5.2.2.4 | 1-byte TCP segments out of order | h,e,l,l,o | hello | hello |
| 5.2.2.5 | 1-byte TCP segments with interleaved invalid sequence numbers | h,e,l,l,o | hello | hello |
| 5.3.2.1 | Invalid IP checksum | hello | <no message> | <no message> |
| 5.3.2.1 | Invalid TCP checksum | hello | <no message> | <no message> |
| 5.3.2.2 | Invalid IP version | hello | <no message> | <no message> |
| 5.3.2.3 | Invalid IP header size | hello | <no message> | <no message> |
| 5.3.2.4 | Invalid TCP header size | hello | <no message> | 8 bytes garbage + hello |

Legend:
, = how string was fragmented/segmented
- = byte from original message not retransmitted
/ = two messages sent (first/second)

**Table 1. Summary of Virtual Protocol Stack evaluation results.**

## 5.1 IP Fragmentation Test

### *5.1.1 Description*

Packet fragmentation involves taking a larger TCP segment and breaking it up into multiple IP packets, when the link layer protocol, for example Ethernet IEEE 802.3, does not support packets over a certain size. For standard Ethernet IEEE 802.3, the maximum transfer unit is 1500 bytes [10][11]. Each operating system handles packet fragmentation differently and must be tested to ensure proper Virtual Protocol Stack construction.

Fragmentation is tested by constructing fragmented packets with offsets that can cause old data to be overwritten or new data to be lost. Overwriting occurs when a fragmented packet overlaps or replaces previously received data that has yet to be processed, as in Figure 11. Normally, data is not processed until it changes into a contiguous format.

An attacker uses IP fragmentation to sneak past a monitoring network intrusion detection system by breaking his/her traffic up into small fragments. Many intrusion detection systems cannot handle fragment reassembly [4][6], so by fragmenting his/her traffic, the intrusion detection system is blind. When an intrusion detection system is capable of reassembling fragmented packets, the attacker can take advantage of a mismatch between the reassembly algorithms used by attempting to overlap fragments.

To accomplish fragment rewrite, the attacker must craft the fragmentation offset fields of the IP header and insure that the TCP checksum corresponds with the desired data. By rewriting fragments, the attacker is attempting an Insertion/Evasion attack where the intrusion detection system receives a message that is different than the one received by the target. The attacker is attempting to insert or delete characters into a message so that it hides its true purpose from the intrusion detection system.

Solaris 2.6 and Microsoft NT 4.0 handle overlapping packet fragments differently than Irix 5.3, 4.4BSD, HP-UX, and Linux. When a data-overlapping packet arrives, Solaris 2.6 and Microsoft NT 4.0 discard new data, while other operating systems would overwrite the data [4]. Virtual Protocol Stacks are constructed to differentiate implementations of packet fragment overlap. Once the target host is tested, a Virtual Protocol Stack processed fragments in exactly the same fashion as the target host, which prevents the attacker from using the intrusion detection's fragment reassembly algorithm against itself.

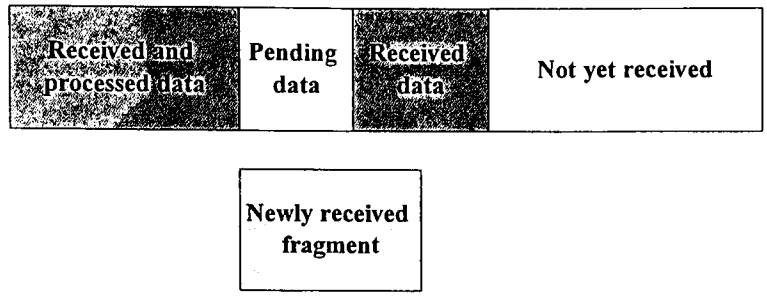


**Figure 11. Illustration of fragment overlap that causes data to be overwritten.**

*5.1.2 Test Procedure*

For each test, an attack character string, in this case the string "hello", was sent to the target host while Virtual Protocol Stacks is monitoring the data traffic. The target machine executed a simple application that prints to the console whatever data it receives over the network. The attacker application sends the known character string, using the method described in the following subsections. The main purpose of these tests was to demonstrate that Virtual Protocol Stacks receive the same message as the target machine. All tests assume complete TCP three-way handshake has taken place [4].

*5.1.2.1 8-byte Fragments in Order*

This test parsed the character string into 8-byte IP fragments and sent them in order to the target machine, where they should be reassembled in the proper order. The TCP segment which was 26 bytes in length (20 byte header + 6 bytes of data), was fragmented into four IP fragments. The first three fragments were each eight bytes in length, while the final segment was two bytes in length. Note that the length of the data in an IP fragment must by a multiple of eight bytes, except for the last fragment [13]. This is a requirement of the IP protocol. This test demonstrated that Virtual Protocol Stacks was capable of handling basic IP fragmentation reassembly.

#### *5.1.2.2 Out of Order Message Using 8-byte Fragments*

To make the task of fragment reassembly more complex, the message is broken into 8-byte IP fragments and transmitted out of order. Out of order means that Virtual Protocol Stacks needed to buffer data before it is completely reassembled. The evaluation tool pre-selected an order for the fragment transmission. Based on the IP fragmentation offsets the order in which they were sent was 3,1,2,0. This test confirms that Virtual Protocol Stacks was capable of buffering fragments and reassembling them into a TCP segment.

#### *5.1.2.3 8-byte Fragmented Message with Rewrite*

This test divided the TCP segment into four IP fragments, three of which are eight bytes of IP data and one that contains two bytes. To perform the rewrite, the evaluation software sent an extra IP fragment in the middle of sending the original IP fragments. This extra IP fragment was designed to rewrite the last two 16-bit words of the TCP header, which includes the TCP checksum, and the first four bytes of data in the TCP segment. The rewrite character string was "xxxx". The Linux system allowed the rewrite to occur and resulted in a string of "xxxxo", where the first four characters where overwritten by the extra IP fragment. The Windows system did not allow the rewrite, and produced the original string "hello". Virtual Protocol Stacks used the results from its test scenarios and produced the correct string for each operating system.

### 5.2 Desynchronizing with Sequence Numbers

#### *5.2.1 Description*

Sequence numbers are used to keep track of each byte of data that is sent over a TCP connection, which enables TCP to provide a mechanism for reliable transportation of data

using retransmission. The desynchronization of sequence numbers in a TCP connection often causes problems within an intrusion detection system [4]. Desynchronization could lead to garbled messages and false connection termination. Sequence numbers are used to inform the sender of various data expected by the receiver and in return, the sender verifies the data received.

Attackers attempt to desynchronize the intrusion detection system by sending packets with invalid sequence numbers. If the network stack for the intrusion detection system resynchronizes itself on false sequence numbers, the system could be blinded by monitoring data in an out of order fashion. To prevent incorrect resynchronization, the Virtual Protocol Stack was constructed so that it handles all sequence number packets in the same fashion as the target host does.

### *5.2.2 Test Procedure*

For the TCP sequence number tests, the evaluation tool sent a known character string using specially crafted TCP segments. The Virtual Protocol Stacks began by evaluating the target host before the tests are executed. The goal was to verify that Virtual Protocol Stacks receive the same message as the target host. The target host ran an application that printed out whatever traffic it receives from the attacker. The main purpose of these tests was to confirm that Virtual Protocol Stacks received the same message as the target machine. All tests assume complete TCP three-way handshake has taken place [4].

#### *5.2.2.1 One Segment Repeat*

Since TCP is a reliable transport protocol that uses retransmission to provide this capability, segments were received that are duplicates of previously received segments. Some intrusion detection systems do blind insertion of data while doing TCP reassembly [4], where data coming over the network is assumed to be in order and the intrusion

detection system reassembles in a first in-first out manner. The test message was broken up into six 1-byte TCP segments, and one segment was selected and sent twice. Virtual Protocol Stacks discarded the repeat in the same manner as the target systems.

#### *5.2.2.2 Segment Rewrite*

To determine that Virtual Protocol Stacks handles data identically to the target host, a 1-byte segment was sent that rewrites a previously received segment. The TCP segments are sent out of order to insure that the data from the TCP segments have been passed on up the protocol stack. Both the Linux and Windows systems discarded the rewrite segment. Virtual Protocol Stacks had identified this characteristic of the Linux and Windows TCP/IP implementation during testing and the Virtual Protocol Stacks' TCP/IP representation of these systems caused it to receive the same message as the two target systems.

#### *5.2.2.3 Segment Overlap with Rewrite*

This test sent the character string using 1-byte TCP segments, but one segment was two bytes in length that attempted to overwrite one of the previously received 1-byte segments. The sequence numbers in the 2-byte segment indicated an overlap. This test illustrates that Virtual Protocol Stacks processed the segment overlap identically as the targets.

The Windows system did not allow the overlap rewrite to occur and produced the original "hello" string. The Linux system allowed the rewrite to occur and produced the string "helmo" where the "m" replaced the original "l". Using the information gathered by Virtual Protocol Stacks testing sequence, Virtual Protocol Stacks was able to handle the differences in the target systems' implementations and receive the correct messages.

#### *5.2.2.4 Send All Segments Out Of Order*

Sending the message segments out of order evaluated Virtual Protocol Stacks ability to reassemble the TCP data stream. For this test, a 1-byte TCP segments was chosen as the initial packet and sent, followed by the next. This illustrates Virtual Protocol Stacks' ability to buffer and reassemble an out of order TCP data stream.

#### *5.2.2.5 Interleave Message with Invalid Sequence Numbers*

To demonstrate Virtual Protocol Stacks' network implementation, segments of a message were transmitted interleaved with segments with invalid sequence numbers. The invalid segments had data that makes erroneous reassembly obvious to the evaluator. Virtual Protocol Stacks handled the inaccurate sequence numbers the equivalent to the target host. For this test, a valid sequence number was incremented by a random number larger than 2000 and smaller than 8191. Since the invalid sequence numbers were outside the range of valid data, Virtual Protocol Stacks and the target receive the identical message with no interference from the invalid sequence numbers.

### 5.3 Malformed TCP and IP Header Options

#### *5.3.1 Description*

Malformed TCP and IP header options and checksums can be constructed in such a way that target hosts reject packets, while intrusion detection systems accept them [4]. In accepting invalid data, the intrusion detection system has become a victim to the insertion attack. The network stacks of the intrusion detection system must be wary of how packets with abnormal header options and checksums are handled. Virtual Protocol Stacks ensures

that the packets received by the intrusion detection system are handled in an identical fashion as the target, by testing different header option and checksum scenarios during stack construction.

### *5.3.2 Test Procedure*

To perceive the effects of malformed TCP and IP headers on packet handling, five tests were run against the target host. The Virtual Protocol Stacks began by evaluating the target host. All tests assumed successful TCP three-way handshake [4].

#### *5.3.2.1 Invalid TCP and IP Checksums*

To mitigate problems due to errors in a message, checksums are used to detect errors in the IP header and TCP segments. To demonstrate that Virtual Protocol Stacks was calculating and verifying the checksums, two test messages were sent, one with an invalid IP checksum and another with an invalid TCP checksum. In both cases, the packets were dropped by Virtual Protocol Stacks, which corresponds with the behavior of the target operating systems.

#### *5.3.2.2 Invalid IP Version*

A Version field in the IP header indicates which version of the IP protocol is contained in the IP packet. Since the Internet runs in an environment that is mainly version 4 of the Internet Protocol, Virtual Protocol Stacks must verify that the protocol that it was working with was correctly identified. The evaluation software sent an IP version of 3 to both target systems. In both cases, the target systems discarded the packets.

#### *5.3.2.3 Invalid IP Header Size*

In the IP header, a field exists that indicates the size of the IP header. The header size is measured in 32-bit words. At a minimum, the header size is five, which is required to deliver a message. If the header size is less than five, the packet should be dropped. Virtual Protocol Stacks verified that the target checks for minimum header size and handled header size identically to the target. The Linux and Windows systems discarded the invalid packets.

#### *5.3.2.4 Invalid TCP Header Size*

The TCP header contains a field indicating the length of the header in 32-bit words. At a minimum, five 32-bit words (or 20 bytes) are needed for a viable header. For the Virtual Protocol Stacks test, a header length of three 32-bit words (or 12 bytes) was placed into the TCP header with six bytes of data. The Linux machine discarded the TCP segment as invalid, while the Windows system accepted the TCP segment and dequeued fourteen bytes of data. The last six bytes of the fourteen was the message data, while the other eight bytes were from the TCP header length, the TCP flags, the TCP window size, the TCP checksum, and the TCP urgent pointer field in the TCP header. From the results of the Virtual Protocol Stacks tests, the TCP representation of the Windows machine accounted for this anomaly and handled the TCP segment in the same manner as the Windows system.

*Chapter 6*

# RELATED WORK

Most network based intrusion detection systems run packet sniffers that receive all packets traveling along the local network segment. Unfortunately, unlike Virtual Protocol Stacks, post-processing of raw data packets is rarely performed and therefore packets are accepted with no accounting for their handling on the target machine.

## 6.1 High Performance Network Intrusion Detection System

In the High-Performance Network Intrusion Detection System [5], fragment packet headers, instead of data, are examined for intrusion signatures by watching for overlapping or smaller fragmented packets than the TCP header [5]. Unfortunately, facilities to handle data examination are non-existent, due to an inability to reassemble IP fragments and TCP segments. In High-Performance Network Intrusion Systems, it assumes that IP fragment should never be smaller than the size of a TCP header, whose smallest size is twenty bytes. Therefore, if a packet is fragmented below this threshold, it is considered an attack.

In the IP fragmentation attack, the attacker attempts to sneak past the intrusion detection system by fragmenting his/her traffic. The system alerts the administrators of packet fragment overlap and fragments below a minimum size, but fails to recognize actual data received by the target host. System administrators connect real-time packet watchers to monitor the actual data being passed, however due to fragment overlap, dropped or overwritten data causes uncertainty in the message contents.

As an intrusion detection system that monitors network traffic, High Performance Network Intrusion Detection System is similar to Virtual Protocol Stacks. However, Virtual Protocol Stacks attempts to make the data source for the intrusion search algorithm more reliable by reassembling IP fragments and TCP segments. The increase in reliability will help decrease the number of false positives created by a strict IP fragment length threshold and increase the intrusion detection capability by reassembling the TCP data stream in a manner that mirrors the target.

## 6.2 Denmac Systems Incorporated Test

In November of 1999, Denmac Systems Incorporated performed a test, based on the tests previously done by Ptacek and Newsham, on four network intrusion systems; ISS RealSecure 3.0, NFR Network Flight Recorder 4.0, Computer Associates Sessionwall-3.3.1, and Axent NetProwler 3.0 [6]. Ptacek and Newsham demonstrated how an intruder could evade an intrusion detection system by fragmenting TCP segments and desynchronizing it with abnormal TCP option flags [4].

When Ptacek and Newsham ran their tests in 1998, all systems failed fragment reassembly [4]. However, in the Denmac test, only NFR Network Flight Recorder managed to detect and identify fragmentation attacks of overlap, out of order reception, and small packet sizes. The most recent version of Network Flight Recorder incorporated changes to handle fragmentation.

On the other hand, ISS RealSecure detected an attack in four of the five fragmentation attacks, but incorrectly identified them. The only attack that ISS RealSecure recognized correctly was partial TCP connection establishment attacks. Unfortunately, ISS RealSecure failed to detect TCP header option and TCP checksum attacks altogether [6].

A drawback to the Denmac test was its system configuration, the only systems used were Microsoft NT 4.0 [6]. Unfortunately, utilizing only one configuration eliminates the ability to see how intrusion detection systems function in heterogeneous environments. NFR Network Flight Recorders may not have handled packet reassembly properly, if the target machine was running the another operating system, such as Linux. As noted by Ptacek and Newsham, Microsoft NT 4.0 handles fragment overlap differently than Linux [4].

In the Denmac Systems, Inc test, current intrusion detection systems have been proven not to handle TCP packet fragmentation. Virtual Protocol Stacks aids intrusion detection systems in identifying fragmentation attacks and providing data to intrusion detection systems of attacker traffic. Various operating systems were be taken into account by Virtual Protocol Stacks when running tests, which allows for heterogeneous environments to be monitored by the same intrusion detection system.

### 6.3 Snort

Similar to the High-Performance Network Intrusion Detection System, Snort does not handle the reassembly of packets. By using a promiscuous network interface device to sniff packets of the local network, it runs a pattern-matching algorithm to determine if any data stream or network packet headers are similar to any known attack. Described by a set of rules, these attack patterns are added manually by the system administrator.

Snort makes an assumption about minimum packet size, and if the packet size is less than 128 bytes, a fragmentation attack alert will be logged [7]. Since Snort logs alerts based on short packets, overlap of fragments does not seem to be covered by Snort. Snort lacks the capability to detect TCP segment overlap, because it does not handle TCP packet stream

reassembly [7] and does not have a method for storing packet history. Therefore, the inability to reassemble packets could cause Insertion/Evasion attack strategies.

Virtual Protocol Stacks processes overlapping packets identical to the target hosts to avoid Insertion/Evasion attacks and furnish reliable network traffic data to the intrusion detection system. Virtual Protocol Stacks could operate as a data source for an intrusion detection system such as Snort, which uses a pattern-matching algorithm on the data for detection. With Virtual Protocol Stacks as a data source, Snort would not be fooled with Insertion/Evasion attack methods.

### 6.4 Bro

Bro is a network-based intrusion detection system designed to handle high throughput environments. To capture packets from the network, Bro uses the libpcap library, also used by such applications as TCPDump and Snort, which filters out undesired packets at the lowest level of the operating system handling network data [9].

Since Bro obtains packets directly from the network, it must complete fragment and TCP stream reassembly to detect and prevent Insertion/Evasion attacks. Packet reassembly is completed by checking for sequence number overlap and data differences [9]. If a packet overlaps a previously received packet, the data should not differ unless the checksum is corrupt. However, if the data does differ, an alert is issued by Bro [9]. Since Bro is unaware of packet handling by the target machine, responses are limited to overlapping data. Alerts can only be issued, but do not provide any data as a clue to the system administrator.

Unlike Bro, the Virtual Protocol Stack's network drivers have the ability to accurately reassemble packet data exactly as the target machine would. Providing the packet data to the system administrator gives a better picture of what is happening on the network.

*Chapter 7*

# FUTURE WORK

Two particular areas exist where Virtual Protocol Stacks must expand future testing into, which are the TCP connection state machine and TCP flags. Once incorporated, Virtual Protocol Stacks will have enough coverage to stave off most Insertion/Evasion attacks. The undefined areas of the TCP/IP specification would by modeled by Virtual Protocol Stacks TCP/IP representations. Virtual Protocol Stacks must be incorporated into an intrusion detection system since it is designed as a tool, and not a complete system. The public domain intrusion detection system Snort offers a possible home for Virtual Protocol Stack's unique capabilities. Snort could benefit from Virtual Protocol Stacks adaptive IP fragmentation reassembly capabilities to improve attack detection against different systems.

## 7.1 TCP Connection State Machine

The TCP connection state machine defines how a TCP connection behaves to certain TCP flag options such as SYN and FIN. The state machine defines their reactions to these options, but does not clearly define reactions to other stimuli such as data in connection establishment packets. The TCP state machine must be analyzed, especially during the connection establishment and termination. During these states, attempts to pass data to that target could be missed or overzealously accepted by the intrusion detection system when the target discards. If the intrusion detection system does the opposite of the target system, the attacker has succeeded. Therefore, the intrusion detection system will have an inaccurate image of network activity, and create a security hole. Virtual Protocol Stacks armed with TCP state machine knowledge will prevent such instances from occurring.

## 7.2 TCP Flags

In the TCP header, flags are used to trigger state transitions in the TCP state machine and provide information about segment contents to the receiver. To leverage Virtual Protocol Stack's adaptive capabilities, the test set used to evaluate the different hosts' reactions to TCP options must be enlarged to encompass abnormal events, such as multiple packets or packets with invalid sequence numbers. Since there are flags in the TCP header that are only used during certain parts of a TCP connection, Virtual Protocol Stacks should test for reactions when these flags are set, in cases when they should not be. An example is the SYN flag, which should only be set during connection establishment. If the flag is set during other states, the data activity is unknown in those packets. Adding these scenarios to Virtual Protocol Stack's test suite will provide better monitoring protection to systems.

## 7.3 Application

Since Virtual Protocol Stacks are designed as a building block for application layer intrusion detection systems, it needs to be built into an intrusion detection system. Virtual Protocol Stacks could ultimately be applied to the Snort intrusion detection system to solidify Snort's IP fragmentation reassembly capabilities. Currently Snort uses the standard libpcap libraries, and with some modifications, it can incorporate IP fragmentation capabilities of Virtual Protocol Stacks. Utilizing the TCP reassembly capabilities of Virtual Protocol Stacks will be more difficult since Snort is not stream based system, but packet based. Once incorporated, Snort's ability to monitor heterogeneous networks would benefit from Virtual Protocol Stack's adaptability to target system.

*Chapter 8*

# CONCLUSION

A common problem that exists today in networks monitored by intrusion detection systems is the ability for Insertion/Evasion attacks to occur from the inconsistencies that exist in the implementations of the TCP/IP protocol. This attack strategy attempts to evade detection by formatting the TCP/IP packets, such that the monitoring system receives a message that differs from the one received by the target. Insertion adds extra characters to the monitor's received message, while Evasion removes characters. In order to compensate for this style of attack, the monitoring system must either retrieve the data from the target or have a model of the system's TCP/IP implementation. Since retrieving data from the target wastes system and network resources, the model approach is ideal.

In a heterogeneous environment with a collection of computers with various architectures and operating systems, models of these systems are difficult to maintain, because of the amount of monitoring effort required. Any patch applied to a system may affect the TCP/IP implementation. To ease the workload, a beneficial monitoring system should be able to automatically adapt to the TCP/IP protocols.

Virtual Protocol Stacks takes the model approach and makes it adaptable to the network environment. To make the system adaptable, Virtual Protocol Stacks runs a series of tests against the target system. From the results of these tests, a representation of the TCP/IP protocol stack is constructed and is used to monitor packets destined for that particular host. This representation is specific to that host and ensures that all TCP/IP packets are processed in the same manner as the host.

By testing the system being monitored, Virtual Protocol Stacks attempts to eliminate the use of the Insertion/Evasion attack. The various testing scenarios executed during Virtual Protocol Stacks evaluation proved to handle the different TCP/IP implementations. In three cases, the two test systems handled TCP/IP packets differently. The two test systems, Linux and Windows, were chosen due to their characteristics, which were the differences in handling TCP segment and IP fragmentation rewrite. In both cases, Virtual Protocol Stacks correctly handled both systems by consistently identifying the test message as received by the target. This demonstrates that capability of Virtual Protocol Stacks to avert Insertion/Evasion attacks using the rewrite approach.

Virtual Protocol Stacks is a building block for application layer intrusion detection systems, for providing a reliable stream of data. By using Virtual Protocol Stacks, a heterogeneous environment will be protected from Insertion/Evasion attacks against the monitoring system by handling the TCP/IP implementation characteristics of the target system. Virtual Protocol Stacks provides data to the application layer intrusion detection system that is consistent with the data received by the target. This similarity prevents Insertion/Evasion attacks from being used against the intrusion detection system. Virtual Protocol Stacks are beneficial to any intrusion detection system that is intending to monitor systems of different types due its ability to adapt to the TCP/IP implementations of each host.

# BIBLIOGRAPHY


[1] R. G. Bace, *Intrusion Detection.* Indianapolis, IN; Macmillian Technical Publishing, 2000.

[2] R. Anderson and A. Khattak, "The Use of Information Retrieval Techniques for Intrusion Detection," Computer Laboratory, University of Cambridge, UK, June 1997.

[3] W. R. Stevens, *TCP/IP Illustrated, Volume 1,* Reading, Massachusetts; Addison-Wesley Publishing Company, 1994.

[4] T. H. Ptacek and T. H. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Inc., January 1998.

[5] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag, "A High-Performance Network Intrusion Detection System," Proceedings of the 6th ACM conference on Computer and Communications Security, 1999, Pages 8 - 17.

[6] Denmac Systems, Inc., "Network Based Intrusion Detection - A Review of Technologies," Denmac Systems, Inc., November 1999.

[7] M. Roesh "Snort - Lightweight Intrusion Detection for Networks," USENIX LISA Conference, November 1999.

[8] CERT "CERT Advisory CA-2000-13 Two Input Validation Problems in FTPD," CERT Coordination Center, November 21, 2000.

[9] V. Paxson "Bro: A System for Detecting Network Intruders in Real-Time," Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998.

[10] A. S. Tanenbaum, *Computer Networks.* Upper Saddle River, NJ; Prentice Hall PTR, 1996.

[11] J. F. Kurose and K. W. Ross, *Computer Networking, A Top-Down Approach Featuring the Internet.* Reading, MA; Addison-Wesley, 2000.

[12] J. Gosling and H. McGilton, "The Java Language Environment, A White Paper," Sun Microsystems, Mountain View, CA, May 1996.

[13] J. Postel, "Internet Protocol: DARPA Internet Program Protocol Specification," RFC 791, September 1981.

[14] The TcpDump Group, http://www.tcpdump.org

[15] M. D. Schiffman, Libnet-1.0,
http://www.packetfactory.net/libnet

[16] R. Braden and D. Borman,
"Computing the Internet Checksum,"
RFC 1071, September 1988.

[17] D. Kramer, "The Java Platform,"
Sun Microsystems, Inc., Mountain View,
CA, May 1996.

[18] J. S. Fritzinger and M. Mueller,
"Java Security," Sun Microsystems, Inc.,
Mountain View CA, 1996.

```
/**************************************************************************
* RCS info
* $Id: ByteUtils.java,v 1.4 2001/03/30 00:44:45 dmcgann Exp $
*
* $Log: ByteUtils.java,v $
* Revision 1.4  2001/03/30 00:44:45  dmcgann
* Added comments.
* Added function to convert int into array of bytes that are
* in network order.
*
* Revision 1.3  2001/03/02 01:54:20  dmcgann
* Added function to convert byte array to long.
*
* Revision 1,2  2001/02/18 01:14:41  dmcgann
* Added Word16 to byte array conversion function.
*
* Revision 1.1  2001/02/07 00:25:25  dmcgann
* Initial revision
*
*
**************************************************************************/

public class ByteUtils
{
/**************************************************************************
* Function converts an array of bytes into a int.
**************************************************************************/
   static int BytesToInt(byte[] Buffer, int Length)
   {
      int i;
      int result = 0;
      int wrap;

      for(i = 0; i < Length; i++)
      {
         wrap = Buffer[i];

         if(wrap < 0)
            wrap += 256;

         if(i == 0)
            result = wrap;
         else
            result += (wrap << (i * 8));
      }

      return(result);
   }

/**************************************************************************
* This function converts an array of bytes into a long.
**************************************************************************/
   static long BytesToLong(byte[] Buffer, int Length)
   {
      int i;
      int result = 0;
      int wrap;

      for(i = 0; i < Length; i++)
      {
         wrap = Buffer[i];

         if(wrap < 0)
            wrap += 256;

         if(i == 0)
            result = wrap;
         else
            result += (wrap << (i * 8));
      }
```





```
      return(result);
   }

/**************************************************************************
* This function converts in int into a 4 byte array.
**************************************************************************/
   static void Word16BitsToBytes(byte[] Buffer, int Input)
   {
      int i;
      int index = 3;

      for(i = 0; i < 4; i++)
         Buffer[index--] = (byte)((Input >> (8 * i)) & 0xFF);
   }

/**************************************************************************
* This function converts in int into a 4 byte array where MSB byte
* goes into last element.
**************************************************************************/
   static void IntToBytes(int Input, byte[] Output)
   {
      Output[3] = (byte)((Input >> 24) & 0xFF);
      Output[2] = (byte)((Input >> 16) & 0xFF);
      Output[1] = (byte)((Input >> 8) & 0xFF);
      Output[0] = (byte)(Input & 0xFF);
   }
}
```

```
/*****************************************************************************
* RCS info
* $Id: Disconnect.java,v 1.2 2001/03/01 01:26:47 dmcgann Exp $
*
* $Log: Disconnect.java,v $
* Revision 1.2  2001/03/01 01:26:47  dmcgann
* Changed ip checksum to 0, which is default for IP and TCP checksum
* calculation by the injector.
*
* Revision 1.1  2001/02/17 21:25:19  dmcgann
* Initial revision
*
*
*****************************************************************************/

public class Disconnect
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =   0x0028;
   private static short   ipId                  =   0x026A;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;
   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x11;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data;
   private static short   dataLength = 0;

   public static int GetPacket(byte[] Buffer)
   {
      int index = 0;
      int j;

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)([ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)[(ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)([ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = [byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
```





```
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);

      return(index);
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }

   public static void SetIpDestAddress(int Dest)
   {
      ipDestAddress = Dest;
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetTcpSrcPort(int SrcPort)
   {
      tcpSrcPort = SrcPort;
   }

   public static void SetTcpSeqNumber(long SequenceNumber)
   {
      tcpSeqNumber = SequenceNumber;
   }

   public static void SetTcpAckNumber(long AckNumber)
   {
      tcpAckNumber = AckNumber;
   }
}
```

```
/*****************************************************************************
* RCS info
* $Id: ethernet.c,v 1.3 2001/02/24 22:29:41 dmcgann Exp $
*
* $Log: ethernet.c,v $
* Revision 1.3  2001/02/24 22:29:41  dmcgann
* Added sending of destination ethernet address to PacketRouter.
*
* Revision 1.2  2001/01/14 21:42:51  dmcgann
* Added RCS identifiers to header comments.
*
*
*****************************************************************************/
#include <stdio.h>

#include "sniffcom.h"


#define SIZE_OF_DEST_ADDR              6
#define SIZE_OF_SRC_ADDR               6
#define SIZE_OF_TYPE_FIELD             2
#define SIZE_OF_DATA                1500

typedef enum
{
   TYPE_IP = 0x0800,
   TYPE_ARP = 0x0806,
   TYPE_RARP = 0x0835,
   TYPE_LAST_ENTRY
} EthernetTypes_t;

typedef struct
{
   char destinationAddr[SIZE_OF_DEST_ADDR];
   char sourceAddr[SIZE_OF_SRC_ADDR];
   char type[SIZE_OF_TYPE_FIELD];
   char Data[SIZE_OF_DATA];
} EthernetPacket_t;


void EthernetProcessor(const char* DataPtr, unsigned int Length)
{
   unsigned          i;
   EthernetPacket_t* packet = (EthernetPacket_t*)DataPtr;
   EthernetTypes_t   packetType;

   if(Length > (SIZE_OF_DATA + SIZE_OF_DEST_ADDR + SIZE_OF_SRC_ADDR +
      SIZE_OF_TYPE_FIELD))
   {
      printf("Length to large\n");
      return;
   }

   printf("Destination address: ");

   for(i = 0; i < SIZE_OF_DEST_ADDR; i++)
   {
      printf("%02X ", (packet->destinationAddr[i] & 0xFF));
   }

   printf("\n");

   printf("Source address: ");

   for(i = 0; i < SIZE_OF_SRC_ADDR; i++)
   {
      printf("%02X ", (packet->sourceAddr[i] & 0xFF));
   }

   printf("\n");
```





```
   printf("Type field: ");

   for(i = 0; i < SIZE_OF_TYPE_FIELD; i++)
      printf("%02X ", (packet->type[i] & 0xFF));

   printf("\n");

   packetType = packet->type[0] << 8;
   packetType |= packet->type[1];
   packetType &= 0xFFFF;

   if(packetType == TYPE_IP)
   {
      SendPacketToPacketRouter(&(packet->destinationAddr),
                               (DataPtr + 14), (Length - 14));
   }
   else
      printf("Invalid type: %04X\n", packetType);

}
```

```
/**************************************************************************
* RCS info
* $Id: HostTcpIpProperties.java,v 1.2 2001/03/30 00:47:56 dmcgann Exp $
*
* $Log: HostTcpIpProperties.java,v $
* Revision 1.2  2001/03/30 00:47:56  dmcgann
* Added comments.
* Added flag for Invalid IP header length.
*
* Revision 1.1  2001/03/11 01:09:18  dmcgann
* Initial revision
*
*
**************************************************************************/

class HostTcpIpProperties
{
   public boolean AcceptsIpFrags          = true;
   public boolean AllowsIpFragRewrite     = false;
   public boolean AcceptInvalidIpVersion  = false;
   public boolean AcceptInvalidIpHeaderLength = false;

   /* 0 - packet rejected, 1 - accepted as is, 2 - from link layer info */
   public byte    IncorrectIpTotalLength  = 0;
   public boolean AcceptInvalidIpChecksum = false;

   /* 0 - packet rejected, 1 - assumes min size, 2 - accepted as is */
   public byte    InvalidTcpHeaderSize    = 0;
   public boolean AllowTcpSeqRewrite      = false;
   public boolean AllowTcpSeqOverlap      = false;
   public boolean AcceptInvalidTcpChecksum = false;

   public String IpAddress;

/**************************************************************************
* Returns the IP of the host that this object represents.
**************************************************************************/
   public HostTcpIpProperties(String IpAddr)
   {
      IpAddress = IpAddr;
   }

/**************************************************************************
* Prints out the information contained in this object.
**************************************************************************/
   public void Print()
   {
      System.out.println("Host IP address   + IpAddress);
      System.out.println("AcceptsIpFrags    + AcceptsIpFrags);
      System.out.println("AllowsIpFragRewrite " + AllowsIpFragRewrite);
      System.out.println("AcceptInvalidIpVersion " + AcceptInvalidIpVersion);
      System.out.println("IncorrectIpTotalLength   + IncorrectIpTotalLength);
      System.out.println("AcceptInvalidIpChecksum " + AcceptInvalidIpChecksum);
      System.out.println("AcceptInvalidIpHeaderLength " +
         AcceptInvalidIpHeaderLength);

      System.out.println("InvalidTcpHeaderSize    + InvalidTcpHeaderSize);
      System.out.println("AllowTcpSeqRewrite " + AllowTcpSeqRewrite);
      System.out.println("AllowTcpSeqOverlap   + AllowTcpSeqOverlap);
      System.out.println("AcceptInvalidTcpChecksum " +
         AcceptInvalidTcpChecksum);
   }
}
```

```
/*****************************************************************************
*
* Description: This injector interface uses the direct link layer interface
*              that libnet provides. This allows for the setting of all
*              IP and TCP options.
*
* RCS info
* $Id: injectorEth.c,v 1.3 2001/03/23 01:26:59 dmcgann Exp $
*
* $Log: injectorEth.c,v $
* Revision 1.3  2001/03/23 01:26:59  dmcgann
* Removed debug statements.
* Added some comments.
*
* Revision 1.2  2001/03/11 01:22:04  dmcgann
* Add support for testng bad checksums.
*
* Revision 1.1  2001/02/24 22:18:21  dmcgann
* Initial revision
*
*
*****************************************************************************/
#include <stdio.h>
#include <libnet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFER_SIZE      2048
#define LISTEN_PORT     15089

typedef enum
{
   FALSE,
   TRUE
} BOOL;

typedef struct
{
   u_char             ipVersion;
   u_char             headerLength;
   u_char             tos;
   unsigned short     totalLength;
   unsigned short     identifier;
   unsigned short     fragFields;
   u_char             ttl;
   u_char             transportProtocol;
   unsigned short     checksum;
   unsigned long      srcIp;
   unsigned long      destIp;
   u_char*            dataPtr;
   unsigned int       dataLength;
} IpPacket_t;

static u_char   PacketBuffer[BUFFER_SIZE];
static u_char   TxPacketBuffer[BUFFER_SIZE];

static u_char   DestEthAddr[6] = {0x00, 0x20, 0x78, 0x03, 0xE6, 0xD2};
static u_char   SrcEthAddr[6] = {0x00, 0x20, 0x78, 0x1D, 0x2A, 0x9C};

/*** Private function prototype ***/
static int ReadLengthFromSocket(int SfD);
static void PrintIpPacket(IpPacket_t* Packet);
static void ParseArrayIntoIpPacket(IpPacket_t* Packet, int Length);


int main(int argc, char* argv[])
{
   int                packetSize;
```





```
   u_char*            packet;
   int                networkSock;
   int                c;
   int                listenSock;
   int                remoteSock;
   struct sockaddr_in serverAddress;
   struct sockaddr_in remoteAddress;
   BOOL               injectorEnabled = TRUE;
   int                packetLength = 0;
   int                packetIndex;
   int                ipPacketLength;
   int                tcpPacketLength;
   IpPacket_t         txPacket;
   int                amount;
   int                remoteLength;
   int                i;
   int                addOffset = 0;

   struct libnet_link_int *LLptr;

   LLptr = libnet_open_link_interface("eth0", PacketBuffer);

   if(LLptr == NULL)
   {
      libnet_error(LIBNET_ERR_FATAL, "libnet open\n");
   }

   /* Construct server socket for the tester to connect to */
   /* to perform packet injection.                         */
   if((listenSock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
      perror("socket");
      return(-1);
   }

   bzero((char*)&serverAddress, sizeof(serverAddress));

   serverAddress.sin_family = AF_INET;
   serverAddress.sin_addr.s_addr = inet_addr("198.100.0.25");
   serverAddress.sin_port = htons(LISTEN_PORT);

   if(bind(listenSock, (struct sockaddr*)&serverAddress,
      sizeof(serverAddress)) < 0)
   {
      perror("bind");
      return(-1);
   }

   listen(listenSock, 1);

   remoteLength = sizeof(remoteAddress);
   remoteSock = accept(listenSock, (struct sockaddr*)&remoteAddress,
      &remoteLength);

   if(remoteSock < 0)
   {
      perror("accept");
      return(-1);
   }

   while(injectorEnabled == TRUE)
   {
      packetLength    = ReadLengthFromSocket(remoteSock);

      packetIndex = 0;

      /* Read in the size of the packet coming form the Tester */
      do
      {
         amount = read(remoteSock, &PacketBuffer[packetIndex],
```

```
      (packetLength - packetIndex));

   packetIndex += amount;

)while((amount > 0) && (packetIndex < packetLength));

ParseArrayIntoIpPacket(&txPacket, packetLength);

PrintIpPacket(&txPacket);

/* sbtract off ethernet address that was sent over with packet */
packetLength -= 6;

libnet_build_ethernet(DestEthAddr, SrcEthAddr,
   ETHERTYPE_IP, NULL, packetLength, TxPacketBuffer);

libnet_build_ip((packetLength - LIBNET_IP_H), /* size of packet */
                txPacket.tos,               /* IP TOS */
                txPacket.identifier,        /* IP ID */
                txPacket.fragFields,        /* fragmentation flags and offset
                txPacket.ttl,               /* TTL */
                txPacket.transportProtocol, /* transport laye protocol */
                txPacket.srcIp,
                txPacket.destIp,
                txPacket.dataPtr,
                (packetLength - LIBNET_IP_H),
                (TxPacketBuffer + ETH_H));

/* Overwrite IP version and header length with value from Tester */
TxPacketBuffer[ETH_H] =
   ((txPacket.ipVersion << 4) | txPacket.headerLength);

/* overwrite IP total length field */
TxPacketBuffer[(ETH_H + 2)] = ((txPacket.totalLength >> 8) & 0xFF);
TxPacketBuffer[(ETH_H + 3)] = (txPacket.totalLength & 0xFF);

/* don't do checksum if recv'd checksum = 0x2222 */
if(txPacket.checksum != 0x2222)
{
   if(libnet_do_checksum((TxPacketBuffer + ETH_H),
      IPPROTO_IP, LIBNET_IP_H) < 0)
   {
      libnet_error(LIBNET_ERR_FATAL, "libnet_do_checksum failed\n");
      return(-1);
   }
}

/* don't do checksum if recv'd checksum = 0x1111 */
if((txPacket.checksum != 0x1111) && (packetLength >= 40) &&
   (txPacket.fragFields == 0))
{
   if(libnet_do_checksum((TxPacketBuffer + ETH_H),
         IPPROTO_TCP, (packetLength - LIBNET_IP_H)) < 0)
   {
      libnet_error(LIBNET_ERR_FATAL, "libnet_do_checksum failed\n");
      return(-1);
   }
}
else
   printf("debug: no checksum calculated as requested\n");

c = libnet_write_link_layer(LLptr, "eth0",  TxPacketBuffer, (packetLength + E
if (c < (packetLength + ETH_H))
{
    libnet_error(LN_ERR_WARNING, "libnet_write_ip only wrote %d bytes\n", c);
}
else
{
    printf("construction and injection completed, wrote all %d bytes\n", c);
}
```





```
    }

    /*
     *  Shut down the interface.
     */
    if(libnet_close_link_interface(LLptr) < 0)
    {
        libnet_error(LN_ERR_WARNING, "libnet_close_raw_sock couldn't close the inte
    }

    return(1);
}

static int ReadLengthFromSocket(int SfD)
{
   int                  tmp;
   unsigned             i;
   int                  value;
   u_char               buffer[4];
   int                  lengthIndex = 3;
   int                  amount = 0;

   do
   {
      amount = read(SfD, &buffer[lengthIndex] , 1);

      lengthIndex -= amount;

   )while((amount > 0) && (lengthIndex >= 0));

   if(lengthIndex < 0)
   {
      for(i = 0; i < 4; i++)
      {
        tmp = buffer[i];

        if(tmp < 0)
           tmp += 256;

        if(i != 0)
           value += (tmp << (i * 8));
        else
           value = tmp;
      }

      printf("debug: Rx length = %d\n", value);
   }

   return(value);
}

static void ParseArrayIntoIpPacket(IpPacket_t* Packet, int Length)
{
   unsigned index = 0;
   int      i;

   for(i = 0; i < 6; i++)
      DestEthAddr[i] = PacketBuffer[index++];

   Packet->headerLength = PacketBuffer[index] & 0xF;
   Packet->ipVersion = (PacketBuffer[index++] >> 4) & 0xF;

   Packet->tos = PacketBuffer[index++];

   Packet->totalLength = (PacketBuffer[index++] << 8);
   Packet->totalLength |= PacketBuffer[index++];

   Packet->identifier = PacketBuffer[index++];
   Packet->identifier |= (PacketBuffer[index++] << 8);
```

```
    Packet->fragFields = PacketBuffer[index++];
    Packet->fragFields |= (PacketBuffer[index++] << 8);

    Packet->ttl = PacketBuffer[index++];

    Packet->transportProtocol = PacketBuffer[index++];

    Packet->checksum = PacketBuffer[index++];
    Packet->checksum |= (PacketBuffer[index++] << 8);

    Packet->srcIp = PacketBuffer[index++];
    Packet->srcIp |= (PacketBuffer[index++] << 8);
    Packet->srcIp |= (PacketBuffer[index++] << 16);
    Packet->srcIp |= (PacketBuffer[index++] << 24);

    Packet->destIp = PacketBuffer[index++];
    Packet->destIp |= (PacketBuffer[index++] << 8);
    Packet->destIp |= (PacketBuffer[index++] << 16);
    Packet->destIp |= (PacketBuffer[index++] << 24);

    Packet->dataPtr = &PacketBuffer[index];
    Packet->dataLength = Length - ((Packet->headerLength * 4) - 6);
}

static void PrintIpPacket(IpPacket_t* Packet)
{
    int i;

    printf("IP version: %X\n", Packet->ipVersion);
    printf("IP header length: %X\n", Packet->headerLength);
    printf("IP TOS: %02X\n", Packet->tos);
    printf("IP total length: %04X\n", Packet->totalLength);
    printf("IP identifier: %02X\n", Packet->identifier);
    printf("IP fragmentation fields: %02X\n", Packet->fragFields);
    printf("IP TTL: %X\n", Packet->ttl);
    printf("IP Transport protocol: %X\n", Packet->transportProtocol);
    printf("IP checksum: %02X\n", Packet->checksum);
    printf("IP src address: %08X\n", Packet->srcIp);
    printf("IP dest address: %08X\n", Packet->destIp);
    printf("IP Length: %02d\n", Packet->dataLength);
}
```

```
/*************************************************************************
* RCS info
* $Id: IpFragData.java,v 1.2 2001/03/23 01:15:17 dmcgann Exp $
*
* $Log: IpFragData.java,v $
* Revision 1.2  2001/03/23 01:15:17  dmcgann
* Added comments.
* Added function to truncate fragments in cases of overlap.
* Added copying of data into object.
*
* Revision 1.1  2001/03/11 01:08:12  dmcgann
* Initial revision
*
*
*************************************************************************/

public class IpFragData
{
   private int    baseIndex;
   private int    fragLength;
   private byte[] data;

/*************************************************************************
* Constructor for IP frag data class. Needs IP frag offset, length of IP
* data, and data itself passed in.
*************************************************************************/
   public IpFragData(int Index, int Length, byte[] Data)
   {
      int i;
      data = new byte[Length];

      baseIndex = Index;
      fragLength = Length / 8;

      for(i = 0; i < Length; i++)
         data[i] = Data[i];
   }

/*************************************************************************
* Function to return the fragments length. 1 count = 8 bytes.
*************************************************************************/
   public int GetFragLength()
   {
      return(fragLength);
   }

/*************************************************************************
* Gets the fragment offsets for this fragment.
*************************************************************************/
   public int GetIndex()
   {
      return(baseIndex);
   }

/*************************************************************************
* Returns the data in this IP fragment.
*************************************************************************/
   public byte[] GetData()
   {
      return(data);
   }

/*************************************************************************
* When having to deal with IP fragmentatipn overlap, we must be able to
* truncate the length of an IP fragment to make room for another IP
* fragment. Num equal to the number of 8 byte chunks that must be discarded.
*************************************************************************/
   public void TruncateFirstXBytes(int Num)
   {
      int i;
```





```
      for(i = 0; i < ((fragLength - Num) * 8); i++)
      {
         data[i] = data[i + Num];
      }

      fragLength -= Num;
      baseIndex += Num;
   }
}
```

```
/*****************************************************************************
* RCS info
* $Id: IpPacket.java,v 1.9 2001/03/30 01:17:09 dmcgann Exp $
*
* $Log: IpPacket.java,v $
* Revision 1.9  2001/03/30 01:17:09  dmcgann
* Added comments.
* Added calculation of IP checksum during construction.
* Fixed bug in SetData() which prevented TCP packet equal to 20 bytes
* from being constructed.
*
* Revision 1.8  2001/03/11 01:10:54  dmcgann
* Added more public functions to retrieve needed header data.
*
* Revision 1.7  2001/03/03 16:13:42  dmcgann
* Fixed bug in creation of TCP packet from data in IP packet. Before was
* checking flags when module needed to check frag fields.
*
* Revision 1.6  2001/03/02 01:53:14  dmcgann
* Added ethernet address and function to support it use.
*
* Revision 1.5  2001/02/18 01:15:47  dmcgann
* Added ability to dynamically set src and destination addresses.
*
* Revision 1.4  2001/02/07 00:27:58  dmcgann
* Added use of TcpPacket class to parse IP data into TCP header.
*
* Revision 1.3  2001/02/03 22:43:13  dmcgann
* Changed source and destination address representations to
* Strings from Integers.
*
* Revision 1.2  2001/01/21 23:28:15  dmcgann
* Added public functions to fetch src and dest address.
* Made all class variables and variable setting functions private.
*
* Revision 1.1  2001/01/21 01:00:29  dmcgann
* Initial revision
*
*****************************************************************************/

class IpPacket
{
   private byte     ipVersion;
   private byte     headerLength;
   private short    typeOfService;
   private int      totalLength;
   private int      identifier;
   private byte     flags;
   private int      fragmentOffset;
   private short    timeToLive;
   private short    protocol;
   private int      checksum;
   private byte[]   sourceAddress = new byte[4];
   private byte[]   destinationAddress = new byte[4];
   private byte[]   data;
   private int      dataLength = 0;
   private boolean  tcpPacketExists = false;

   private TcpPacket   TcpContents;

   private String sourceInetAddress;
   private String destinationInetAddress;

   private byte[]   destEthAddr = new byte[6];

/*****************************************************************************
* Function calculates the IP checksum and returns TRUE if packet is valid,
* else FALSE in returned.
*****************************************************************************/
```





```
public boolean ChecksumValid()
{
   int  sum = 0;
   int  carry;

   sum += (ipVersion << 12);
   sum += (headerLength << 8);
   sum += typeOfService;

   sum += totalLength;

   sum += identifier;

   sum += (flags << 13);
   sum += fragmentOffset;

   sum += (timeToLive << 8);
   sum += protocol;

   sum += checksum;

   if(sourceAddress[0] < 0)
      sum += (sourceAddress[0] + 256);
   else
      sum += sourceAddress[0];

   if(sourceAddress[1] < 0)
      sum += ((sourceAddress[1] + 256) << 8);
   else
      sum += (sourceAddress[1] << 8);

   if(sourceAddress[2] < 0)
      sum += (sourceAddress[2] + 256);
   else
      sum += sourceAddress[2];

   if(sourceAddress[3] < 0)
      sum += ((sourceAddress[3] + 256) << 8);
   else
      sum += (sourceAddress[3] << 8);

   if(destinationAddress[0] < 0)
      sum += (destinationAddress[0] + 256);
   else
      sum += destinationAddress[0];

   if(destinationAddress[1] < 0)
      sum += ((destinationAddress[1] + 256) << 8);
   else
      sum += (destinationAddress[1] << 8);

   if(destinationAddress[2] < 0)
      sum += (destinationAddress[2] + 256);
   else
      sum += destinationAddress[2];

   if(destinationAddress[3] < 0)
      sum += ((destinationAddress[3] + 256) << 8);
   else
      sum += (destinationAddress[3] << 8);

   carry = (sum >> 16);
   sum &= 0xffff;
   sum += carry;
   sum &= 0xffff;

   if(sum != 0xFFFF)
      return(false);
   else
      return(true);
```

```
   }

/**************************************************************************
* Returns the header length of the IP packet.
**************************************************************************/
   public int GetHeaderLength()
   {
      return(headerLength);
   }

/**************************************************************************
* Returns the identifier of the IP packet.
**************************************************************************/
   public int GetId()
   {
      return(identifier);
   }

/**************************************************************************
* Returns the length of the data contained in the IP packet.
**************************************************************************/
   public int GetDataLength()
   {
      return(dataLength);
   }

/**************************************************************************
* Returns the data contained in this IP packet.
**************************************************************************/
   public byte[] GetData()
   {
      return(data);
   }

/**************************************************************************
* Returns the total length of the IP packet indiacted by the header.
**************************************************************************/
   public int GetIpTotalLengthField()
   {
      return(totalLength);
   }

/**************************************************************************
* Gets the length of the packet based of header length plus the length of
* data.
**************************************************************************/
   public int GetPacketLength()
   {
      return((headerLength * 4) + dataLength);
   }

/**************************************************************************
* Returns the IP version entry in the header.
**************************************************************************/
   public byte GetIpVersion()
   {
      return(ipVersion);
   }

/**************************************************************************
* Returns the fragmentation flags.
**************************************************************************/
   public byte GetFragmentFlags()
   {
      return(flags);
   }

/**************************************************************************
* Returns the number in the Fragmentation offset field.
**************************************************************************/
```





```
   public int GetFragmentOffset()
   {
      return(fragmentOffset);
   }

/**************************************************************************
* Returns the Ethernet address of the destination host for this IP packet.
**************************************************************************/
   public long GetDestinationEthernetAddr()
   {
     return(ByteUtils.BytesToLong(destEthAddr, 6));
   }

/**************************************************************************
* Returns the Ethernet address of the destination host for this IP packet.
**************************************************************************/
   public byte[] GetDestinationEthernetAddrBytes()
   {
     return(destEthAddr);
   }

/**************************************************************************
* Returns the IP address of the source for this IP packet.
**************************************************************************/
   public int GetSourceHostAddressInt()
   {
     return(ByteUtils.BytesToInt(sourceAddress, 4));
   }

/**************************************************************************
* Returns the IP address of the destination for this IP packet.
**************************************************************************/
   public int GetDestinationHostAddressInt()
   {
     return(ByteUtils.BytesToInt(destinationAddress, 4));
   }

/**************************************************************************
* Returns the IP address of the destination for this IP packet.
**************************************************************************/
   public String GetDestinationHostAddress()
   {
     return(destinationInetAddress);
   }

/**************************************************************************
* Returns the IP address of the source for this IP packet.
**************************************************************************/
   public String GetSourceHostAddress()
   {
     return(sourceInetAddress);
   }

/**************************************************************************
* Returns TRUE if this IP packet contains a TCP segment.
**************************************************************************/
   public boolean IsProtocolTcp()
   {
      if(protocol == 6)
         return(true);
      else
         return(false);
   }

/**************************************************************************
* Returns the TCP contents of this IP packet.
**************************************************************************/
   public TcpPacket GetTcpPacket()
   {
       return(TcpContents);
```

```
   }

/**************************************************************************
* Constructor for IP packet.
**************************************************************************/
   public IpPacket(byte[] InputByteStream, int Length)
   {
      byte[] tmp = new byte[4];
      int    index = 0;
      int    i;
      int    hl;


      if(Length < 26)
      {
         System.out.println("Invalid Length:   + Length);
         return;
      }

      for(i = 0; i < 6; i++)
         destEthAddr[i] = InputByteStream[index++];

      this.SetPacketVersion(InputByteStream[index]);
      this.SetHeaderLength(InputByteStream[index++]);
      this.SetTypeOfService(InputByteStream[index++]);

      tmp[0] = InputByteStream[index++];
      tmp[1] = InputByteStream[index++];

      this.SetTotalLength(tmp);

      tmp[0] = InputByteStream[index++];
      tmp[1] = InputByteStream[index++];

      this.SetIdentifier(tmp);

      this.SetFlags(InputByteStream[index]);

      tmp[0] = InputByteStream[index++];
      tmp[1] = InputByteStream[index++];

      this.SetFragmentOffset(tmp);

      this.SetTimeToLive(InputByteStream[index++]);
      this.SetProtocol(InputByteStream[index++]);

      tmp[0] = InputByteStream[index++];
      tmp[1] = InputByteStream[index++];

      this.SetChecksum(tmp);

      tmp[0] = InputByteStream[index++];
      tmp[1] = InputByteStream[index++];
      tmp[2] = InputByteStream[index++];
      tmp[3] = InputByteStream[index++];

      this.SetSourceAddress(tmp);

      tmp[0] = InputByteStream[index++];
      tmp[1] = InputByteStream[index++];
      tmp[2] = InputByteStream[index++];
      tmp[3] = InputByteStream[index++];

      this.SetDestintionAddress(tmp);

      if(headerLength < 5)
         hl = 5;
      else
         hl = headerLength;
```





```
      this.SetData(InputByteStream, Length, ((hl * 4) + 6));
   }

/**************************************************************************
* Prints information about IP packet.
**************************************************************************/
   public void Print()
   {
      System.out.println("IP header ->");
      System.out.println("Version: " + Integer.toHexString(ipVersion));
      System.out.println("Header Length: " + headerLength);

      System.out.println("Type Of Service:   +
         Integer.toHexString(typeOfService));

      System.out.println("Total Length:   + totalLength);
      System.out.println("Identifier: " + Integer.toHexString(identifier));
      System.out.println("Flags: " + Integer.toHexString(flags));
      System.out.println("Fragment Offset: " + fragmentOffset);
      System.out.println("Time To Live: " + timeToLive);
      System.out.println("Protocol: " + Integer.toHexString(protocol));
      System.out.println("Checksum:   + Integer.toHexString(checksum));

      System.out.println("Source address:   + sourceInetAddress.toString());

      System.out.println("Destination address:   +
         destinationInetAddress.toString());

      System.out.println("TCP header ->");

      if(tcpPacketExists == true)
         TcpContents.Print();
   }

/**************************************************************************
*************                Private functions                *************
**************************************************************************/
   private void SetPacketVersion(byte Version)
   {
      ipVersion = (byte)((Version >> 4) & 0xF);
   }

   private void SetHeaderLength(byte HeaderLength)
   {
      headerLength = (byte)(HeaderLength & 0xF);
   }

   private void SetTypeOfService(byte TypeOfService)
   {
      typeOfService = (short)TypeOfService;

      if(typeOfService < 0)
         typeOfService += 256;
   }

   private void SetTotalLength(byte[] TotalLength)
   {
      totalLength = (int)((TotalLength[0] << 8) + (int)(TotalLength[1] & 0xFF));
      totalLength &= 0xFFFF;
   }

   private void SetIdentifier(byte[] Identifier)
   {
      identifier = (int)((Identifier[0] << 8) + (int)(Identifier[1] & 0xFF));
      identifier &= 0xFFFF;
   }

   private void SetFlags(byte Flags)
   {
      flags = (byte)((Flags >> 5) & 0x7);
```

```
    }

    private void SetFragmentOffset(byte[] FragmentOffset)
    {
        fragmentOffset = ((FragmentOffset[0] << 8) & 0x1F) +
            (int)(FragmentOffset[1] & 0xFF);
    }

    private void SetTimeToLive(byte TimeToLive)
    {
        timeToLive = TimeToLive;

        if(timeToLive < 0)
            timeToLive += 256;
    }

    private void SetProtocol(byte Protocol)
    {
        protocol = Protocol;

        if(protocol < 0)
            protocol += 256;
    }

    private void SetChecksum(byte[] Checksum)
    {
        checksum = (Checksum[0] << 8) + (int)(Checksum[1] & 0xFF);
        checksum &= 0xFFFF;
    }

    private void SetSourceAddress(byte[] SourceAddress)
    {
      short i;
      short[] temp = new short[4];

      for(i = 0; i < 4; i++)
      {
          temp[i] = SourceAddress[i];
          sourceAddress[(3 - i)] = SourceAddress[i];

          if(temp[i] < 0)
              temp[i] += 256;

      }

      String stringAddress = new String(temp[0] +    ' + temp[1] +  .  +
          temp[2] + "." + temp[3]);

      sourceInetAddress = stringAddress;
    }

    private void SetDestintionAddress(byte[] DestinationAddress)
    {
      short i;
      short[] temp = new short[4];

      for(i = 0; i < 4; i++)
      {
          destinationAddress[(3 - i)] = DestinationAddress[i];
          temp[i] = DestinationAddress[i];

          if(temp[i] < 0)
              temp[i] += 256;

      }

      String stringAddress = new String(temp[0] +      + temp[1] + '.' +
          temp[2] + "." + temp[3]);

      destinationInetAddress = stringAddress;
```





```
    }

    private void SetData(byte[] Data, int Length, int Offset)
    {
        int i;
        data = new byte[(Length - Offset)];

        dataLength = Length - Offset;

        for(i = 0; i < (Length - Offset); i++)
            data[i] = Data[i + Offset];

        if((dataLength >= 20) && (fragmentOffset == 0) && (protocol == 6))
        {
            TcpContents = new TcpPacket(data, dataLength);
            TcpContents.SetSrcAndDestIpAddress(sourceAddress, destinationAddress);
            tcpPacketExists = true;
        }
    }
}
```

```
/*************************************************************************
* RCS info
* $Id: IpProcessor.java,v 1.2 2001/03/30 01:53:25 dmcgann Exp $
*
* $Log: IpProcessor.java,v $
* Revision 1.2  2001/03/30 01:53:25  dmcgann
* Added comments.
* Fixed TCP checksum calculation problem after IP fragments were reassembled.
*
* Revision 1.1  2001/03/11 01:11:48  dmcgann
* Initial revision
*
*
*************************************************************************/
import java.lang.*;
import java.util.*;

class IpProcessor extends Thread
{
   private LinkedList          RecvdPacketBuffer = new LinkedList();
   private HostTcpIpProperties Properties;
   private String              IpAddress;

   private ArrayList           TcpProcessorList = new ArrayList();

   private LinkedList          FragmentIndexBuffer = new LinkedList();
   private int                 FragId;
   private int                 SegLength;

/*************************************************************************
* Constructor
*************************************************************************/
   public IpProcessor(HostTcpIpProperties Props)
   {
      Properties = Props;
   }

/*************************************************************************
* Returns the IP address of the Host Properties object.
*************************************************************************/
   public String GetHostIpAddr()
   {
      return(Properties.IpAddress);
   }

/*************************************************************************
* This function is used by the PacketRouter to hand off packets to the
* IP Processor.
*************************************************************************/
   public synchronized void InformIpPacketFromHost(IpPacket Packet)
   {
      RecvdPacketBuffer.add((Object)Packet);

      this.notify();
   }

/*************************************************************************
* Thread main loop.
*************************************************************************/
   public void run()
   {
      while(true)
      {
         IpPacket packet;

         packet = CheckPacketBuffer();

         ProcessIpPacket(packet);
      }
   }
```





```
/*************************************************************************
* Method checks for a packet from the PacketRouter.
*************************************************************************/
   private synchronized IpPacket CheckPacketBuffer()
   {
      while(RecvdPacketBuffer.isEmpty() == true)
      {
         try
         {
            wait();
         }
         catch(InterruptedException e)
         {
           System.out.println(e.toString());
         }
      }

      return((IpPacket)RecvdPacketBuffer.removeFirst());
   }

/*************************************************************************
* This method checks for IP fragmented packets. If the packet is not
* fragmented, the packet is handed off to the TCP processor based on the
* destination port. If it is fragmented, the packet is passed to the
* fragment reassemby function.
*************************************************************************/
   private void ProcessIpPacket(IpPacket Packet)
   {
      if(CheckPacket(Packet) == false)
         return;

      if((Packet.GetFragmentOffset() == 0) &&
         ((Packet.GetFragmentFlags() & (0x1)) == 0))
      {
         TcpPacket tcpPacket = Packet.GetTcpPacket();

         if(tcpPacket != null)
         {
            TcpProcessor tcpProc;

            tcpProc = FindTcpProcessor(tcpPacket.GetDestinationPort());

            tcpProc.InformTcpPacketFromHost(tcpPacket);
         }
         else
         {
            System.out.println("IpProcessor: null TCP packet");
            Packet.Print();
         }
      }
      else
      {
         System.out.println("IpProcessor: fragment IP packet found - >   +
            Packet.GetFragmentFlags());

         ProcessFragmentedPacket(Packet);
      }
   }

/*************************************************************************
* Method searches TCP Processor list for processor to send TCP packet to.
*************************************************************************/
   private TcpProcessor FindTcpProcessor(int Port)
   {
      TcpProcessor tcpProc = null;
      int          i;

      for(i = 0; i < TcpProcessorList.size(); i++)
      {
```

```
         tcpProc = (TcpProcessor)TcpProcessorList.get(i);

         if(tcpProc.Port == Port)
         {
            break;
         }
      }

      if(i == TcpProcessorList.size())
      {
         tcpProc = new TcpProcessor(Port, Properties);
         tcpProc.start();
         TcpProcessorList.add((Object)tcpProc);

         System.out.println("IpProcessor: created TCP proc for   + Port);
      }

      return(tcpProc);
   }

/**************************************************************************
* This function checks the validity of the packet based on the Host
* characteristics derived from the Tester.
**************************************************************************/
   private boolean CheckPacket(IpPacket Packet)
   {
      boolean validPacket = true;

      if(Packet.ChecksumValid() == false)
         return(false);

      /* Check for invalid IP header length */
      if((Packet.GetHeaderLength() < 5) || (Packet.GetHeaderLength() > 8))
      {
         validPacket = Properties.AcceptInvalidIpHeaderLength;
      }

      /* Check for acception of non Ipv4 packets */
      if((Packet.GetIpVersion() != 4) &&
         (Properties.AcceptInvalidIpVersion == false))
      {
         validPacket = false;
      }

      /* check for acceptance of fragmented packets */
      if((Packet.GetFragmentOffset() != 0) &&
         (Properties.AcceptsIpFrags == false))
      {
         validPacket = false;
      }

      /* check for acceptance of fragmented packets */
      if(((Packet.GetFragmentFlags() & (~0x2)) != 0) &&
         (Properties.AcceptsIpFrags == false))
      {
         validPacket = false;
      }

      /* reject packet if ip hedaer total length field is not equal */
      /* to the length indicated by the link layer.                 */
      if((Packet.GetIpTotalLengthField() != Packet.GetPacketLength()) &&
         (Properties.IncorrectIpTotalLength  == 0))
      {
         validPacket = false;
      }

      return(validPacket);
   }

/**************************************************************************
```





```
* This method handles fragment reassembly.
**************************************************************************/
   private void ProcessFragmentedPacket(IpPacket Packet)
   {
      int     i;
      boolean inserted = false;

      if(FragmentIndexBuffer.isEmpty() == true)
      {
         /* this is the first packet */
         FragId = Packet.GetId();
         SegLength = -1;

         IpFragData ipFrag = new IpFragData(Packet.GetFragmentOffset(),
            Packet.GetDataLength(), Packet.GetData());

         FragmentIndexBuffer.addFirst((Object)ipFrag);

         if(Packet.GetFragmentFlags() == 0)
         {
            SegLength = Packet.GetFragmentOffset() +
               Packet.GetDataLength() / 8;
         }
      }
      else
      {
         /* check to see if fragment belongs */
         if(FragId != Packet.GetId())
         {
            System.out.println("ID mismath, packet tossed");
            return;
         }

         if((Packet.GetFragmentFlags() == 0) && (SegLength < 0))
         {
            SegLength = Packet.GetFragmentOffset() +
               Packet.GetDataLength() / 8;
         }

         i = 0;

         while((i < FragmentIndexBuffer.size()) && (inserted == false))
         {
            IpFragData searchFrag = (IpFragData)FragmentIndexBuffer.get(i);

            if(Packet.GetFragmentOffset() < searchFrag.GetIndex())
            {
               inserted = true;

               if((Packet.GetFragmentOffset() + (Packet.GetDataLength() / 8)) <=
                  searchFrag.GetIndex())
               {
                  IpFragData ipFrag = new IpFragData(Packet.GetFragmentOffset(),
                     Packet.GetDataLength(), Packet.GetData());

                  FragmentIndexBuffer.add(i, (Object)ipFrag);
               }
               else if(Properties.AllowsIpFragRewrite == true)
               {

                  int numToTruc = Packet.GetFragmentOffset() +
                     (Packet.GetDataLength() / 8) - searchFrag.GetIndex();

                  System.out.println("Overlap by   + numToTruc);

                  searchFrag.TruncateFirstXBytes(numToTruc);

                  FragmentIndexBuffer.set(i, (Object)searchFrag);

                  IpFragData ipFrag = new IpFragData(Packet.GetFragmentOffset(),
```

```
                Packet.GetDataLength(), Packet.GetData());

            FragmentIndexBuffer.add(i, (Object)ipFrag);
        }
    }
    else if(Packet.GetFragmentOffset() == searchFrag.GetIndex())
    {
        inserted = true;

        if(Properties.AllowsIpFragRewrite == true)
        {
            IpFragData ipFrag = new IpFragData(Packet.GetFragmentOffset(),
                Packet.GetDataLength(), Packet.GetData());

            FragmentIndexBuffer.set(i, (Object)ipFrag);
        }
    }
    else if(Packet.GetFragmentOffset() > searchFrag.GetIndex())
    {

    }


    i++;
}

if(inserted == false)
{
    IpFragData ipFrag = new IpFragData(Packet.GetFragmentOffset(),
        Packet.GetDataLength(), Packet.GetData());

    FragmentIndexBuffer.addLast((Object)ipFrag);
}

boolean complete = true;
int     accum = 0;

if(SegLength > 0)
{
    i = 0;

    while((i < FragmentIndexBuffer.size()) && (complete == true))
    {
        IpFragData searchFrag = (IpFragData)FragmentIndexBuffer.get(i);

        if(accum == searchFrag.GetIndex())
        {
            accum += searchFrag.GetFragLength();
        }
        else
        {
            complete = false;
        }

        i++;
    }

    if((complete == true) && (SegLength == accum))
    {
        int    index = 0;
        byte[] buffer = new byte[256];

        int j = 0;

        while(FragmentIndexBuffer.isEmpty() == false)
        {
            IpFragData pullFrag =
                (IpFragData)FragmentIndexBuffer.removeFirst();

            byte[] tmp = pullFrag.GetData();
```





```
                for(i = 0; i < tmp.length; i++)
                    buffer[index++] = tmp[i];

                j++;
            }

            TcpPacket    tcpPacket = new TcpPacket(buffer, index);

            byte[] sAddr = new byte[4];
            byte[] dAddr = new byte[4];
            int srcAddr  = Packet.GetSourceHostAddressInt();
            int destAddr = Packet.GetDestinationHostAddressInt();

            ByteUtils.IntToBytes(srcAddr, sAddr);
            ByteUtils.IntToBytes(destAddr, dAddr);

            tcpPacket.SetSrcAndDestIpAddress(sAddr, dAddr);

            //tcpPacket.Print();

            TcpProcessor tcpProc;

            tcpProc = FindTcpProcessor(tcpPacket.GetDestinationPort());

            tcpProc.InformTcpPacketFromHost(tcpPacket);
        }
    }
  }
}
```

```
/*****************************************************************************
* RCS info
* $Id: Mobile.java,v 1.3 2001/03/30 01:22:37 dmcgann Exp $
*
* $Log: Mobile.java,v $
* Revision 1.3  2001/03/30 01:22:37  dmcgann
* Added comments.
*
* Revision 1.2  2001/02/18 01:16:54  dmcgann
* Added ability to send results of tests to Tester.
*
* Revision 1.1  2001/01/28 00:46:47  dmcgann
* Initial revision
*
*
*****************************************************************************/

import java.lang.*;
import java.io.*;
import java.net.*;

public class Mobile implements java.io.Serializable
{
   private static Socket CCsocket;
   private static InputStream CCinput;
   private static OutputStream CCoutput;

   private static Socket TestSocket;

   private byte[] RxTestMessageBuffer = new byte[1024];
   private int    RxTestMesgLength = 0;

/*****************************************************************************
* Main function
*****************************************************************************/
   public void StartTest()
   {
      int         command;
      boolean     sessionComplete = false;

      /* Establish command and control connectio with */
      /* VPS. Mobile is server size.                  */
      EnableCCInterface();

      do
      {
         command = GetCommand();

         System.out.println("Received:   " + command);

         switch(command)
         {
            case -1:
               /* error has occurred in get command function */
               SendCommandAcknowledgement();
               System.out.println("error command");
               sessionComplete = true;
               break;

            case 0:
               /* tester has told us to terminate session */
               SendCommandAcknowledgement();
               DisableTestIntarface();
               sessionComplete = true;
               break;

            case 2:
                /* test informing to prepare for test cycle */
                SendCommandAcknowledgement();
                EnableTestInterface();
```





```
                StoreIncomingData();
                break;

            case 3:
                /* send recived message to tester */
                SendCommandAcknowledgement();
                SendRecvdTestMessage();
                break;

            default:
                SendCommandAcknowledgement();
                sessionComplete = true;
                System.out.println("Unexpected case");
                break;
         }

      }while(sessionComplete == false);

      DisableCCInterface();
   }

/*****************************************************************************
* Starts up the Command & Control server interface.
*****************************************************************************/
   private void EnableCCInterface()
   {
      try
      {
         ServerSocket serverSock = new ServerSocket(15087);

         CCsocket = serverSock.accept();

         System.out.println("Mobile CC connection established: " +
            CCsocket.toString());

         serverSock.close();

         CCinput = CCsocket.getInputStream();
         CCoutput = CCsocket.getOutputStream();
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }
   }

/*****************************************************************************
* Reads commands from the CC tester interface.
*****************************************************************************/
   private int GetCommand()
   {
      int command = 0;

      try
      {
         command = CCinput.read();
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }

      return(command);
   }

/*****************************************************************************
* Sends an acknowledgment of command to VPS tester.
*****************************************************************************/
   private void SendCommandAcknowledgement()
   {
```

```
      try
      {
         CCoutput.write(1);
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }
   }

/*****************************************************************************
* Terminates the Command & Control server.
*****************************************************************************/
   private void DisableCCInterface()
   {
      try
      {
         CCinput.close();
         CCoutput.close();
         CCsocket.close();
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }
   }

/*****************************************************************************
* Starts the Tester server interface.
*****************************************************************************/
   private void EnableTestInterface()
   {
      try
      {
         ServerSocket testServer = new ServerSocket(15088);

         TestSocket = testServer.accept();

         System.out.println("Mobile Test connection established:   +
            TestSocket.toString());

         testServer.close();
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }
   }

/*****************************************************************************
* Reads data from the Tester interface until connection is terminated.
*****************************************************************************/
   private void StoreIncomingData()
   {
      try
      {
         int amount;
         int index = 0;
         int i;

         InputStream testInput = TestSocket.getInputStream();

         do
         {
            amount = testInput.read(RxTestMessageBuffer, index,
               (1024 - index));

            System.out.println("Amount ->   + amount);

            if(amount > 0)
```





```
               index += amount;

            } while((amount > 0) && (index < 1024));

            RxTestMesgLength = index;

            String mesg = new String(RxTestMessageBuffer, 0, index);

            System.out.print("Rx mesg -> ' + mesg);

            testInput.close();
         }
         catch(IOException e)
         {
            System.out.println(e.toString());
         }
      }

/*****************************************************************************
* This function sends the buffered data back to the VPS tester via the
* CC interface.
*****************************************************************************/
      private void SendRecvdTestMessage()
      {
         try
         {
            byte length = (byte)(RxTestMesgLength & 0xFF);
            CCoutput.write(length);

            CCoutput.write(RxTestMessageBuffer, 0, RxTestMesgLength);

            CCoutput.flush();
         }
         catch(IOException e)
         {
            System.out.println(e.toString());
         }
      }

/*****************************************************************************
* This function disables the tester interface.
*****************************************************************************/
      private void DisableTestInterface()
      {
         try
         {
            TestSocket.close();
         }
         catch(IOException e)
         {
            System.out.println(e.toString());
         }
      }
   }
```

```
/*****************************************************************************
* RCS info
* $Id: PacketRouter.java,v 1.11 2001/03/23 01:07:38 dmcgann Exp $
*
* $Log: PacketRouter.java,v $
* Revision 1.11  2001/03/23 01:07:38  dmcgann
* Added comments.
* Removed some debug messages.
*
* Revision 1.10  2001/03/11 01:19:02  dmcgann
* Added sending of packets to IpProcessor of known hosts,
*
* Revision 1.9  2001/03/02 01:55:26  dmcgann
* Changed what it sent to Tester when informing it of a target to test, The
* entire IP packet is passed to the Tester instead of only ip address.
*
* Revision 1.8  2001/02/18 01:21:43  dmcgann
* Commented out debug print statements.
*
* Revision 1.7  2001/02/07 00:31:09  dmcgann
* Added capability to pass packets from host under test to Tester.
* Added check for termination of sniffer connection.
*
* Revision 1.6  2001/02/03 22:40:07  dmcgann
* Fixed broken pipe bug caused by byte overflow.
* Changed address representations to Strings instead of Integers.
* Rearranged checking of tcp protocol and local source in else
* statement to prevent null pointer exception.
*
* Revision 1.5  2001/02/03 01:10:11  dmcgann
* Coverted input stream to buffered input stream.
*
* Revision 1.4  2001/01/25 01:44:13  dmcgann
* Added calls to Tester to spawn thread and insert unknown host ip addresses.
*
* Revision 1.3  2001/01/21 23:33:20  dmcgann
* Module now loops on input from sniffer components.
* Filters out packets locally generated,
* Adds new IP destinations to KnownHostsList for testing.
*
* Revision 1.2  2001/01/14 21:41:17  dmcgann
* Fixed RCS Id field in header comments,
*
* Revision 1.1  2001/01/14 21:32:50  dmcgann
* Initial revision
*
*
*****************************************************************************/
import java.net.*;
import java.lang.*;
import java.io.*;
import java.util.*;

class PacketRouter
{
   private static LinkedList ListOfKnownHosts   = new LinkedList();
   private static ArrayList  ListOfIpProcessors = new ArrayList();

/*****************************************************************************
* This function adds an IP processor to the list oh known host with IP
* Processor representations.
*****************************************************************************/
   public static synchronized void AddIpProcessor(IpProcessor ipProcessor)
   {
      ListOfIpProcessors.add((Object)ipProcessor);
      System.out.println("Added proc to ip proc list");
   }

/*****************************************************************************
* This function is used to search the IP Processor list how an IP Processor
```





```
* that corresponds with the passed in IP address.
*****************************************************************************/
   public static synchronized IpProcessor CheckForIpProcessor(String ipAddr)
   {
      int         i;
      IpProcessor ipProc = null;
      boolean     found = false;

      for(i = 0; i < ListOfIpProcessors.size(); i++)
      {
        ipProc = (IpProcessor)ListOfIpProcessors.get(i);

        if(ipAddr.equals(ipProc.GetHostIpAddr()) == true)
        {
           found = true;
           break;
        }
      }

      if(found == false)
         return(null);
      else
         return(ipProc);
   }

/*****************************************************************************
* Main loop for Virtual Protocol Stacks.
*****************************************************************************/
   public static void main(String[] args)
   {
      try
      {
         boolean noErrors = true;
         short   j;
         short[] temp = new short[4];

         /* Get local address for later filtering of packets. */
         InetAddress localAddr = InetAddress.getLocalHost();
         byte[] localAddrByte = localAddr.getAddress();

         for(j = 0; j < 4; j++)
         {
            temp[j] = localAddrByte[j];

            if(temp[j] < 0)
               temp[j] += 256;
         }

         String localAddress = new String(temp[0] + '   + temp[1] +   " +
            temp[2] + "." + temp[3]);

         /* Construct server socket for Packet Sniffer to connect */
         /* to and provide packets to VPS engine.                 */
         ServerSocket sock = new ServerSocket(7770);

         Socket remoteSock = sock.accept();

         BufferedInputStream input =
            new BufferedInputStream(remoteSock.getInputStream(), 3000);

         /* Instatiate the Tester and start its thread running */
         Tester VpsTester = new Tester();
         VpsTester.start();

         do
         {
            byte[] length = new byte[4];
            byte[] data = new byte[2048];
            int    result;
            int    i;
```

```
int    realLength;
int    dataOffset;
int    amount;
int    wrap;

realLength = 0;
dataOffset = 0;
amount = 0;

/* Read in the size of the packet coming across socket */
/* from Packet sniffer. First four bytes for an int    */
/* that indicates the size.                            */
do
{
   result = input.read(length, amount, 1);

   wrap = length[amount];

   if(wrap < 0)
      wrap += 256;

   if(amount > 0)
      realLength += (wrap * (amount * 256));
   else
      realLength += wrap;

   amount += result;

} while(result > 0 && amount < 4);


/* Validate received size */
if(amount == 4)
{
   if(realLength > 2048)
   {
      System.out.println("recv'd length too large");
      realLength = 1;
      noErrors = false;
   }
}
else
{
   if(result < 0)
   {
      System.out.println(
         "sniffer connection terminated...exiting");
      noErrors = false;
   }
   else
   {
      System.out.println("read to many/few bytes:   + result);
      noErrors = false;
   }
}

while((realLength > 0) && (result >= 0))
{
   result = input.read(data, dataOffset, realLength);

   realLength -= result;
   dataOffset += result;
}

if((result >= 0) && (noErrors == true))
{
   IpProcessor ipProc = null;
   IpPacket    ipPacket = new IpPacket(data, dataOffset);
   String      destAddress = ipPacket.GetDestinationHostAddress();
   String      srcAddress = ipPacket.GetSourceHostAddress();
```





```
            /*** NOTE: will need to add rule to eliminate locally */
            /*** sourced traffic.                                 */
            if((ipPacket.IsProtocolTcp() == true) &&
               /* (srcAddress.equals(localAddress) == false) && */
               (destAddress.equals(localAddress) == false))
            {
               ipProc = CheckForIpProcessor(destAddress);

               if(ipProc != null)
               {
                  ipProc.InformIpPacketFromHost(ipPacket);
               }
               else if(ListOfKnownHosts.contains((Object)destAddress) == true)
               {
                  /* do nothing, we know about this host    */
                  /* has already been added to tester list. */
               }
               else
               {
                  /* Found host to evaluate, add to Tester list. */
                  VpsTester.TesterAddHostToTest(ipPacket);

                  System.out.println("debug: Adding -> " + destAddress);

                  ListOfKnownHosts.add((Object)destAddress);
               }
            }
            else
            {
               if(ipPacket.IsProtocolTcp() == false)
               {
                  /* System.out.println("debug: not tcp packet"); */
               }
               else if(destAddress.equals(localAddress) == true)
               {
                  if(VpsTester.IsHostUnderTest(srcAddress) == true)
                  {
                     /* Tell tester we have recived a packet */
                     /* from the Host undet Test.            */
                     VpsTester.InformIpPacketFromHUT(ipPacket);
                  }
               }
            }
         }
      } while(noErrors == true);

      input.close();
      remoteSock.close();
   }
   catch(IOException e)
   {
      System.out.println(e.toString());
   }
  }
}
```

```
/**************************************************************************
* RCS info
* $Id: Recvr.java,v 1.3 2001/03/30 01:13:10 dmcgann Exp $
*
* $Log: Recvr.java,v $
* Revision 1.3  2001/03/30 01:13:10  dmcgann
* Added comments.
* Changed download protocol, first byte indicates length of packet
* following.
*
*
**************************************************************************/
import java.io.*;
import java.net.*;
import java.lang.*;
import java.lang.reflect.*;

class Recvr
{
    public static void main(String[] args)
    {
      try
      {
         ServerSocket sock = new ServerSocket(17770);

         while(true)
         {
            int amount = 0;
            Socket remoteSock = sock.accept();
            int    blockSize;
            byte[] buffer = new byte[127];
            int    offset;

            System.out.println("Connected:   + remoteSock.toString());

            InputStream input = remoteSock.getInputStream();
            OutputStream output = remoteSock.getOutputStream();

            FileOutputStream fOutput = new FileOutputStream("Mobile.class");

            /* first byte indicates length of data packet coming */
            blockSize = input.read();

            System.out.println("Waiting for   + blockSize +   bytes");

            while(blockSize > 0)
            {
               do
               {
                  amount = input.read(buffer, 0, blockSize);

                  blockSize -= amount;

                  if(amount > 0)
                     fOutput.write(buffer, 0, amount);

               } while(blockSize > 0);

               /* first byte indicates length of data packet coming */
               blockSize = input.read();

               System.out.println("Waiting for   + blockSize +   bytes");
            }

            fOutput.flush();

            System.out.println("File flushed");

            fOutput.close();
```





```
            try
            {
               Class mobileClass = Class.forName("Mobile");
               Method[] mobileMethods = mobileClass.getDeclaredMethods();

               try
               {
                  try
                  {
                     Object mobile = mobileClass.newInstance();

                     try
                     {
                        output.write(1);

                        mobileMethods[0].invoke(mobile, null);
                     }
                     catch(InvocationTargetException e)
                     {
                        System.out.println(e.toString());
                     }
                  }
                  catch(IllegalAccessException e)
                  {
                     System.out.println(e.toString());
                  }
               }
               catch(InstantiationException e)
               {
                  System.out.println(e.toString());
               }
            }
            catch(ClassNotFoundException e)
            {
               System.out.println(e.toString());
            }

            input.close();
            output.close();
            remoteSock.close();

            System.out.println("Connection closed");

         }
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }
   }
}
```

```
/*****************************************************************************
* RCS info
* $Id: sniffcom.c,v 1.3 2001/02/24 22:27:58 dmcgann Exp $
*
* $Log: sniffcom.c,v $
* Revision 1.3  2001/02/24 22:27:58  dmcgann
* Added sending of destination ethernet address to PacketRouter.
*
* Revision 1.2  2001/01/25 01:41:31  dmcgann
* Modules now send entire data packet to VPS.
*
* Revision 1.1  2001/01/14 21:43:12  dmcgann
* Initial revision
*
*
*****************************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>

#define ETHER_ADDR_LENGTH        6
static int SocketFd;

/*****************************************************************************
*
*****************************************************************************/
int EnableSniffcom(unsigned Port)
{
   struct sockaddr_in     serverAddr;
   char                   hostName[128];
   struct hostent*        hostEntryPtr;
   int                    result;

   result = gethostname(hostName, sizeof(hostName));

   if(result < 0)
   {
      perror("gethostname");
      return(0);
   }

   hostEntryPtr = gethostbyname(hostName);

   if(hostEntryPtr == NULL)
   {
      herror("gethostbyname");
      return(0);
   }

   bzero(&serverAddr, sizeof(serverAddr));

   serverAddr.sin_family = AF_INET;
   /* serverAddr.sin_addr.s_addr = inet_addr(hostEntryPtr->h_addr); */
   serverAddr.sin_addr.s_addr = inet_addr("198.100.0.25");
   serverAddr.sin_port = htons(Port);

   SocketFd = socket(AF_INET, SOCK_STREAM, 0);

   if(SocketFd < 0)
   {
      perror("socket");
      return(0);
   }

   result = connect(SocketFd, (struct sockaddr*)&serverAddr,
      sizeof(serverAddr));
```





```
   if(result < 0)
   {
      perror("connect");
      return(0);
   }

   return(1);
}

/*****************************************************************************
*
*****************************************************************************/
void SendPacketToPacketRouter(char* DestEthAddr, char* PacketPtr,
   unsigned int Length)
{
   Length += ETHER_ADDR_LENGTH;

   printf("Length id %d\n\n", Length);

   write(SocketFd, (void*) &Length, sizeof(Length));

   write(SocketFd, (void*) DestEthAddr, ETHER_ADDR_LENGTH);

   write(SocketFd, (void*) PacketPtr, (Length - ETHER_ADDR_LENGTH));
}

/*****************************************************************************
*
*****************************************************************************/
void DisableSniffcom(void)
{
   close(SocketFd);
}
```

```
/*****************************************************************************
* RCS info
* $Id: sniffer.c,v 1.4 2001/01/25 01:38:28 dmcgann Exp $
*
* $Log: sniffer.c,v $
* Revision 1.4  2001/01/25 01:38:28  dmcgann
* Sniffer loops until signals are sent to terminate.
* Fixed typo in error message for EnableSniffcom().
*
* Revision 1.3  2001/01/21 01:03:34  dmcgann
* Added catching of signals to gracefully shutdown application.
*
* Revision 1.2  2001/01/14 21:44:12  dmcgann
* Added RCS identifiers to header comments.
* Added calls to sniffcom module for communications with Java.
*
*
*****************************************************************************/
#include <stdio.h>
#include <pcap.h>
#include <signal.h>

#include "ethernet.h"
#include "sniffcom.h"

typedef enum
{
   FALSE = 0,
   TRUE = 1
} BOOL;

#define CAPTURE_LENGTH     1500
#define READ_TIMEOUT        200

/* 0 - loops forever */
#define PACKETS_TO_LOOP       1
#define LOOP_COUNT            1

static pcap_t*  OpenDevice;
static BOOL     SnifferEnabled;

/******** Private function prototypes ********/
static void DisableSniffer(void);
static void SignalHandler(int SignalNumber);
static void PrintPcapError(char* Buffer);

/*****************************************************************************
*
*****************************************************************************/
static void SignalHandler(int SignalNumber)
{
   switch(SignalNumber)
   {
      case SIGHUP:
         printf("caught SIGHUP\n");
         DisableSniffcom();
         DisableSniffer();
         break;

      case SIGINT:
         printf("caught SIGINT\n");
         DisableSniffcom();
         DisableSniffer();
         break;

      case SIGTERM:
         printf("caught SIGTERM\n");
         DisableSniffcom();
         DisableSniffer();
         break;
```





```
      case SIGQUIT:
         printf("caught SIGQUIT\n");
         DisableSniffcom();
         DisableSniffer();
         break;

      default:
         break;
   }
}

/*****************************************************************************
*
*****************************************************************************/
static void PrintPcapError(char* Buffer)
{
   unsigned i;

   for(i = 0; i < PCAP_ERRBUF_SIZE; i++)
      printf("%c", Buffer[i]);

   printf("\n");
}

/*****************************************************************************
*
*****************************************************************************/
static void PcapCallback(u_char* a , const struct pcap_pkthdr* Header,
   const u_char* DataPtr)
{
   unsigned length;
   unsigned capturedLength;
   unsigned i;

   capturedLength = Header->caplen;
   length = Header->len;

   if(length != capturedLength)
      printf("mismatch captured length and length\n");

   /* for(i = 0; i < capturedLength; i++)
      printf("%04d: %02X\n", i, *(DataPtr+i)); */

    EthernetProcessor(DataPtr, capturedLength);
}

/*****************************************************************************
*
*****************************************************************************/
static void DisableSniffer(void)
{
   struct pcap_stat p;

   if(pcap_stats(OpenDevice, &p) >= 0)
   {
      printf("Number packets received: %d\n", p.ps_recv);
      printf("Number packets dropped: %d\n", p.ps_drop);
   }
   else
      printf("No pcap stats\n");

   SnifferEnabled = FALSE;
   pcap_close(OpenDevice);
}

/*****************************************************************************
*
*****************************************************************************/
int main(void)
```

```
{
   char*        device;
   char         errorBuffer[PCAP_ERRBUF_SIZE];
   int          loopResults;
   char         userBuffer[CAPTURE_LENGTH];
   unsigned     i;
   BOOL         success;

   bzero(errorBuffer, sizeof(errorBuffer));

   device = pcap_lookupdev(errorBuffer);

   if(device == NULL)
   {
      printf("pcap lookup error: ");
      PrintPcapError(errorBuffer);
      return(0);
   }

   OpenDevice = pcap_open_live(device, CAPTURE_LENGTH, TRUE, READ_TIMEOUT,
      errorBuffer);

   if(OpenDevice == NULL)
   {
      printf("pcap live open: ");
      PrintPcapError(errorBuffer);
      return(0);
   }

   SnifferEnabled = TRUE;

   if(signal(SIGHUP, SignalHandler) == SIG_ERR)
      perror("signal SIGHUP");

   if(signal(SIGINT, SignalHandler) == SIG_ERR)
      perror("signal SIGINT");

   if(signal(SIGTERM, SignalHandler) == SIG_ERR)
      perror("signal SIGTREM");

   if(signal(SIGQUIT, SignalHandler) == SIG_ERR)
      perror("signal SIGQUIT");

   success = EnableSniffcom(7770);

   if(success == FALSE)
   {
      printf("Failed to establish connection to Packet Router\n");
      DisableSniffer();
   }

   i = 0;

   do
   {

      loopResults = pcap_dispatch(OpenDevice, PACKETS_TO_LOOP, PcapCallback,
         userBuffer);

   } while((loopResults > 0) && (SnifferEnabled));

   if((loopResults < 0) && (SnifferEnabled == TRUE))
      pcap_perror(OpenDevice, "pcap loop");

   if(SnifferEnabled == TRUE)
      pcap_close(OpenDevice);

   DisableSniffcom();

   return(1);
```





```
}
```

```
/**************************************************************************
* RCS info
* $Id: TcpData.java,v 1.1 2001/03/11 01:12:51 dmcgann Exp $
*
* $Log: TcpData.java,v $
* Revision 1.1  2001/03/11 01:12:51  dmcgann
* Initial revision
*
*
**************************************************************************/

public class TcpData
{
   public byte    data;
   public long    sequenceNumber;

   public int compare(TcpData object)
   {
      if(object.sequenceNumber == sequenceNumber)
         return(0);
      else if(object.sequenceNumber > sequenceNumber)
         return(1);
      else
         return(-1);
   }
}
```

```
/**************************************************************************
* RCS info
* $Id: TcpPacket.java,v 1.4 2001/03/30 01:14:58 dmcgann Exp $
*
* $Log: TcpPacket.java,v $
* Revision 1.4  2001/03/30 01:14:58  dmcgann
* Added comments.
* Added calculation of TCP checksum.
* Changed calculation of TCP data length.
*
* Revision 1.3  2001/03/11 01:15:51  dmcgann
* Fixed bug in calculation of length of data.
* Added functions to retrieve information about TCP packet.
*
* Revision 1.2  2001/02/18 01:20:27  dmcgann
* Added many functions for getting information about packet.
*
* Revision 1.1  2001/02/07 00:26:52  dmcgann
* Initial revision
*
*
**************************************************************************/

class TcpPacket
{
   private int    sourcePort;
   private int    destPort;
   private long   seqNumber;
   private long   ackNumber;
   private byte   headerLength;
   private byte   flags;
   private int    windowSize;
   private int    checksum;
   private int    urgentPtr;
   private byte[] data;
   private int    calcdChecksum;

   private short[]   srcIpAddress = new short[4];
   private short[]   destIpAddress = new short[4];

/**************************************************************************
* Constructor for TCP packet class.
**************************************************************************/
   public TcpPacket(byte[] Buffer, int Length)
   {
      /* if length less than 20 bytes, not   */
      /* enough fata to construct TCP header */
      if(Length >= 20)
      {
         byte[] temp = new byte[4];
         int    index = 0;
         int    i;
         int    hi = 0;
         int    lo = 0;

         /* calculate part of the TCP checksum hera */
         calcdChecksum = Length;
         calcdChecksum += 6;  /* protocol type */

         for(i = 0; i < Length; i += 2)
         {
            if(Buffer[i] < 0)
               lo = (Buffer[i] + 256);
            else
               lo = Buffer[i];

            if((i + 1) >= Length)
               hi = 0;
            else
            {
```





```
               if(Buffer[(i + 1)] < 0)
                  hi = (Buffer[(i + 1)] + 256);
               else
                  hi = Buffer[(i + 1)];
            }

            calcdChecksum += ((lo << 8) + hi);
         }

         temp[1] = Buffer[index++];
         temp[0] = Buffer[index++];

         sourcePort = ByteUtils.BytesToInt(temp, 2);

         temp[1] = Buffer[index++];
         temp[0] = Buffer[index++];

         destPort = ByteUtils.BytesToInt(temp, 2);

         temp[3] = Buffer[index++];
         temp[2] = Buffer[index++];
         temp[1] = Buffer[index++];
         temp[0] = Buffer[index++];

         seqNumber = ByteUtils.BytesToInt(temp, 4);

         temp[3] = Buffer[index++];
         temp[2] = Buffer[index++];
         temp[1] = Buffer[index++];
         temp[0] = Buffer[index++];

         ackNumber = ByteUtils.BytesToInt(temp, 4);

         headerLength = (byte)((Buffer[index++] >> 4) & 0xF);

         flags = (byte)(Buffer[index++] & 0x3F);

         temp[1] = Buffer[index++];
         temp[0] = Buffer[index++];

         windowSize = ByteUtils.BytesToInt(temp, 2);

         temp[1] = Buffer[index++];
         temp[0] = Buffer[index++];

         checksum = ByteUtils.BytesToInt(temp, 2);

         temp[1] = Buffer[index++];
         temp[0] = Buffer[index++];

         urgentPtr = ByteUtils.BytesToInt(temp, 2);

         /* index += (headerLength * 4 - 20); */
         index = headerLength * 4;

         data = new byte[(Length - index)];

         for(i = 0; i < (Length - index); i++)
            data[i] = Buffer[(index + i)];
      }
      else
         System.out.println("Tcp PAcket length too small");
   }

/**************************************************************************
* Returns the ACK number contained in the TCP header.
**************************************************************************/
   public long GetAckNumber()
   {
      return(ackNumber);
```

```
   }

/**************************************************************************
* Returns the sequence number contained in the TCP header.
**************************************************************************/
   public long GetSequenceNumber()
   {
      return(seqNumber);
   }

/**************************************************************************
* Returns the binary flags in the TCP header.
**************************************************************************/
   public byte GetFlags()
   {
      return(flags);
   }

/**************************************************************************
* Returns the source port of this packet.
**************************************************************************/
   public int GetSourcePort()
   {
      return(sourcePort);
   }

/**************************************************************************
* Returns the destination port of this packet.
**************************************************************************/
   public int GetDestinationPort()
   {
      return(destPort);
   }

/**************************************************************************
* Returns the header length indicated by the TCP header.
**************************************************************************/
   public byte GetHeaderLength()
   {
      return(headerLength);
   }

/**************************************************************************
* Returns the amount of data contained in this TCP segment.
**************************************************************************/
   public int GetNumberDataBytes()
   {
      return(data.length);
   }

/**************************************************************************
* Returns the data in this TCP segment.
**************************************************************************/
   public byte[] GetData()
   {
      return(data);
   }

/**************************************************************************
* Sets the source and destination addresses for this TCP packet. This is
* needed to calculate the TCP checksum.
**************************************************************************/
   public void SetSrcAndDestIpAddress(byte[] Src, byte[] Dest)
   {
      int    i;
      int    hi = 0;
      int    lo = 0;

      for(i = 0; i < 4; i++)
      {
```





```
         srcIpAddress[i] = Src[i];

         if(srcIpAddress[i] < 0)
            srcIpAddress[i] += 256;

         destIpAddress[i] = Dest[i];

         if(destIpAddress[i] < 0)
            destIpAddress[i] += 256;
      }

      for(i = 0; i < 4; i += 2)
      {
         calcdChecksum += ((srcIpAddress[(i + 1)] << 8) + srcIpAddress[i]);
         calcdChecksum += ((destIpAddress[(i + 1)] << 8) + destIpAddress[i]);
      }

      hi = calcdChecksum >> 16;
      calcdChecksum &= 0xFFFF;
      calcdChecksum += hi;
      calcdChecksum &= 0xFFFF;
   }

/**************************************************************************
* Returns the calculated checksum. Should be 0xFFFF for valid packet.
**************************************************************************/
   public int GetCalculatedChecksum()
   {
      return(calcdChecksum);
   }

/**************************************************************************
* Prints the information contained in the TCP packet object.
**************************************************************************/
   public void Print()
   {
      System.out.println("Source Port: " + sourcePort);
      System.out.println("Dest Port: " + destPort);
      System.out.println("Seq Number: " + seqNumber);
      System.out.println("Ack Number: " + ackNumber);
      System.out.println("Header Length: " + headerLength);
      System.out.println("Flags: " + Integer.toHexString((int)flags));
      System.out.println("Window Size: " + windowSize);
      System.out.println("Checksum: " + Integer.toHexString(checksum));
      System.out.println("Urgent Ptr: " + urgentPtr);
      System.out.println("data length: " + data.length);
      System.out.println("Src addr: " + srcIpAddress[0]);
      System.out.println("Dest addr: " + destIpAddress[0]);
   }
}
```

```
/*****************************************************************************
* RCS info
* $Id: TcpProcessor.java,v 1.2 2001/03/30 01:52:00 dmcgann Exp $
*
* $Log: TcpProcessor.java,v $
* Revision 1.2  2001/03/30 01:52:00  dmcgann
* Added comments.
* Added packet filter based on source port we are handling.
*
* Revision 1.1  2001/03/11 01:14:19  dmcgann
* Initial revision
*
*
*****************************************************************************/
import java.lang.*;
import java.util.*;
import java.io.*;

public class TcpProcessor extends Thread
{
   private LinkedList          RecvdPacketBuffer = new LinkedList();
   private HostTcpIpProperties Properties;
   public  int                 Port;
   private FileOutputStream    Output;
   private long                SequenceNumber = 0;
   private LinkedList          RecvdBytes = new LinkedList();

   private int                 TxPort = 0;

/*****************************************************************************
* Class constructor
*****************************************************************************/
   public TcpProcessor(int DestPort, HostTcpIpProperties Props)
   {
      Properties = Props;
      Port = DestPort;
   }

/*****************************************************************************
* Method used by the IP Processor to pass packets to the TCP processor.
*****************************************************************************/
   public synchronized void InformTcpPacketFromHost(TcpPacket Packet)
   {
      int size;

      RecvdPacketBuffer.add((Object)Packet);

      this.notify();
   }

/*****************************************************************************
* thread main loop.
*****************************************************************************/
   public void run()
   {
      String aPort = "./log/" + Integer.toString(Port);
      int    tcpState = 0;

      System.out.println("TcpProcessor: opening   + aPort);

      try
      {
         Output = new FileOutputStream(aPort);

         while(true)
         {
            TcpPacket packet;
            byte      flags;
            long       seqNumber;
```





```
            packet = CheckPacketBuffer();

            System.out.println(Port + ": TcpState =   + tcpState);

            if((TxPort != packet.GetSourcePort()) && (tcpState > 0))
            {
               System.out.println("Unwanted from ' + packet.GetSourcePort() +
                  " while expecting " + TxPort);
               continue;
            }

            switch(tcpState)
            {
               case 0: /* waiting for sync */
                  flags = packet.GetFlags();

                  if(flags == 0x2)
                  {
                     System.out.println("State =   + tcpState + ' got sync");
                     tcpState = 1;
                     SequenceNumber = packet.GetSequenceNumber();
                     TxPort = packet.GetSourcePort();
                  }
                  break;

               case 1: /* waiting for sync + ack */
                  flags = packet.GetFlags();
                  seqNumber = packet.GetSequenceNumber() - SequenceNumber;

                  if(((flags == 0x10) || (flags == 0x12)) &&
                     (seqNumber == 1))
                  {
                     SequenceNumber = packet.GetSequenceNumber();
                     System.out.println("State = " + tcpState + " got sync ack");

                     tcpState = 2;

                     SequenceNumber = packet.GetSequenceNumber();
                     String mesg = new String();
                     mesg = "<Sync'd with " + packet.GetSourcePort() + ">\n";

                     try
                     {
                        Output.write(mesg.getBytes());
                     }
                     catch(IOException e)
                     {
                        System.out.println(e.toString());
                     }
                  }
                  else if(flags == 0x02)
                  {
                     System.out.println("State = ' + tcpState +
                        " waiting ack but got sync");
                     SequenceNumber = packet.GetSequenceNumber();
                  }
                  else
                     System.out.println("Fell thru");
                  break;

               case 2; /* estabilised */

                  //packet.Print();

                  if(ValidatePacket(packet) == true)
                     ProcessData(packet);

                  flags = packet.GetFlags();

                  System.out.println("Flags -> ' + Integer.toHexString(flags));
```

```
                if((flags == 0x11) || (flags == 0x01))
                {
                   tcpState = 0;
                   System.out.println("Found EOM");

                   /* clear out reassembly buffer */
                   RecvdBytes.clear();

                   String mesg = new String();
                   mesg = "\n<FIN received>\n";

                   try
                   {
                      Output.write(mesg.getBytes());
                   }
                   catch(IOException e)
                   {
                      System.out.println(e.toString());
                   }
                }
                break;

             case 3: /* waitng for fin + ack */
                break;

             default:
                System.out.println("TcpProcessor: Invalid state");
                break;
          }
       }
    }
    catch(FileNotFoundException e)
    {
       System.out.println(e.toString());
    }
 }

/**************************************************************************
* Checks packet buffer for packets from the IP Processor.
**************************************************************************/
 private synchronized TcpPacket CheckPacketBuffer()
 {
    TcpPacket tmp;

    while(RecvdPacketBuffer.size() == 0)
    {
       try
       {
          wait();
       }
       catch(InterruptedException e)
       {
         System.out.println(e.toString());
       }
    }

    tmp = (TcpPacket)RecvdPacketBuffer.removeFirst();
    return(tmp);
 }

/**************************************************************************
* Checks the TCP Packets against the charateristics derived by the Tester.
**************************************************************************/
 private boolean ValidatePacket(TcpPacket Packet)
 {
    boolean validPacket = true;

    if((Packet.GetCalculatedChecksum() != 0xFFFF) &&
       (Properties.AcceptInvalidTcpChecksum  == false))
```





```
    {
       System.out.println("TcpProcessor: Found invalid TCP checksum");
       return(false);
    }

    /* drop packet do to invalid tcp header size */
    if((Packet.GetHeaderLength() < 5) &&
       (Properties.InvalidTcpHeaderSize == 0))
    {
       System.out.println("TcpProcessor: Invalid header size detected");
       return(false);
    }

    return(validPacket);
 }

/**************************************************************************
* This method handles the reassembly for TCP segments into a stream. This
* stream is written out to a log file for evaluation.
**************************************************************************/
 private void ProcessData(TcpPacket Packet)
 {
    long    seqNumber = Packet.GetSequenceNumber();
    int     numBytes  = Packet.GetNumberDataBytes();
    int     i;
    TcpData tcpData;
    int     j;
    boolean inserted;
    byte[]  mesg = Packet.GetData();
    boolean skipSearch = false;
    boolean byteWasInserted = false;

    //Packet.Print();

    if(SequenceNumber > seqNumber)
    {
       System.out.println("Data already processed, skipping.");
       return;
    }

    for(i = 0; i < numBytes; i++)
    {
      tcpData = new TcpData();

      tcpData.data = mesg[i];
      tcpData.sequenceNumber = seqNumber + i;

      inserted = false;
      j = 0;

      while((j < RecvdBytes.size()) && (inserted == false) &&
            (skipSearch == false))
      {
         TcpData dataCheck = (TcpData)RecvdBytes.get(j);

         if(tcpData.compare(dataCheck) > 0)
         {
            RecvdBytes.add(j, (Object)tcpData);
            inserted = true;
            byteWasInserted = true;
         }
         else if(tcpData.compare(dataCheck) == 0)
         {
            inserted = true;

            if(Properties.AllowTcpSeqRewrite == true)
            {
               RecvdBytes.set(j, (Object)tcpData);
            }
            else if ((byteWasInserted == true) &&
```

```
                    (Properties.AllowTcpSeqOverlap == true))
            {
               RecvdBytes.set(j, (Object)tcpData);
            }
         }

         j++;
      }

      if(inserted == false)
      {
         RecvdBytes.add((Object)tcpData);
         skipSearch = true;
         byteWasInserted = true;
      }
   }

   if(RecvdBytes.isEmpty() == false)
   {
      TcpData firstByte = (TcpData)RecvdBytes.getFirst();

      System.out.println("Out Rx: " + firstByte.sequenceNumber +
         " expected: " + SequenceNumber);

      /* see if any contigous data is available to be written */
      /* out to log file                                      */
      while(firstByte.sequenceNumber == SequenceNumber)
      {
         SequenceNumber++;

         try
         {
            String hex = Integer.toHexString((firstByte.data & 0xFF));

            Output.write(hex.getBytes());
            Output.write(' ');
         }
         catch(IOException e)
         {
            System.out.println(e.toString());
         }

         firstByte = (TcpData)RecvdBytes.removeFirst();

         if(RecvdBytes.isEmpty() == false)
         {
            firstByte = (TcpData)RecvdBytes.getFirst();
            System.out.println("In Rx: " + firstByte.sequenceNumber +
               " expected: " + SequenceNumber);
         }
      }
   }
}
}
```

```
/*****************************************************************************
*
* Description: This function transmits a TCP segment using IP fragmentation.
*
* RCS info
* $Id: Test1.java,v 1.2 2001/03/02 01:52:12 dmcgann Exp $
*
* $Log: Test1.java,v $
* Revision 1.2  2001/03/02 01:52:12  dmcgann
* Fixed ip total length field.
*
* Revision 1.1  2001/03/01 00:31:34  dmcgann
* Initial revision
*
*
*****************************************************************************/
import java.util.*;
import java.lang.*;

public class Test1
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =       24;
   private static short   ipId                  =   0x3014;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;
   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x18;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x68, 0x65, 0x6C, 0x6C,
                                  0x6F, 0x20, 0x20, 0x65,
                                  0x66, 0x67, 0x68, 0x69,
                                  0x70, 0x71, 0x72, 0x73,
                                  0x74, 0x75, 0x76, 0x77,
                                  0x78, 0x79, 0x21, 0x22};

   private static short   dataLength = 6;
   private static short   fragIndex = 0;
   private static boolean checksumCalcd = false;

   public static void ResetTest()
   {
      Random rand = new Random();

      ipId = (short)rand.nextInt(65535);

      tcpChecksum = 0;
      fragIndex = 0;
      ipFrag = (short)0x2000;
      ipTotalLength = 20 + 8;
      checksumCalcd = false;
   }

   public static int NumPacketsInTest()
   {
      return(dataLength);
   }
```





```
   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(data);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int    index = 0;
      int    j;

      if(checksumCalcd == false)
      {
         checksumCalcd = true;
         CalcTcpChecksum();
      }

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      if(fragIndex == 0)
      {
         ipFrag += 1;

         Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
         Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpDestPort & 0xFF);
         Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
         Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
         Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);
      }
      else if(fragIndex == 1)
      {
         ipFrag += 1;

         Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
         Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
         Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
         Buffer[index++] = tcpHeaderLength;
         Buffer[index++] = tcpFlags;
         Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      }
```

```
    else if(fragIndex == 2)
    {
       ipFrag += 1;

       Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
       Buffer[index++] = (byte)(tcpChecksum & 0xFF);
       Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
       Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
       Buffer[index++] = data[0];
       Buffer[index++] = data[1];
       Buffer[index++] = data[2];
       Buffer[index++] = data[3];
    }
    else if(fragIndex == 3)
    {
       ipFrag += 1;

       Buffer[index++] = data[4];
       Buffer[index++] = data[5];
       Buffer[index++] = data[6];
       Buffer[index++] = data[7];
       Buffer[index++] = data[8];
       Buffer[index++] = data[9];
       Buffer[index++] = data[10];
       Buffer[index++] = data[11];
    }
    else if(fragIndex == 4)
    {
       ipFrag += 1;
       ipFrag -= 0x2000;
       ipTotalLength = 20 + 4;

       Buffer[index++] = data[12];
       Buffer[index++] = data[13];
       Buffer[index++] = data[14];
       Buffer[index++] = data[15];
       Buffer[index++] = data[16];
       Buffer[index++] = data[17];
       Buffer[index++] = data[18];
       Buffer[index++] = data[19];
    }
    else
    {
       Buffer[index++] = data[20];
       Buffer[index++] = data[21];
       Buffer[index++] = data[22];
       Buffer[index++] = data[23];
    }

    fragIndex++;
    return(index);
}

private static void CalcTcpChecksum()
{
    byte[] buffer = new byte[128];
    int    index = 0;
    int    size;
    int    i;
    int    hi = 0;
    int    lo = 0;
    int    tmp;

    buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpSrcPort & 0xFF);
    buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpDestPort & 0xFF);
    buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
    buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
    buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
```





```
    buffer[index++] = (byte)(tcpSeqNumber & 0xFF);

    buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
    buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
    buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpAckNumber & 0xFF);
    buffer[index++] = tcpHeaderLength;
    buffer[index++] = tcpFlags;
    buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpWinSize & 0xFF);
    buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpChecksum & 0xFF);
    buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
    buffer[index++] = data[0];
    buffer[index++] = data[1];
    buffer[index++] = data[2];
    buffer[index++] = data[3];

    buffer[index++] = data[4];
    buffer[index++] = data[5];
    buffer[index++] = data[6];
    buffer[index++] = data[7];
    buffer[index++] = data[8];
    buffer[index++] = data[9];
    buffer[index++] = data[10];
    buffer[index++] = data[11];

    buffer[index++] = data[12];
    buffer[index++] = data[13];
    buffer[index++] = data[14];
    buffer[index++] = data[15];
    buffer[index++] = data[16];
    buffer[index++] = data[17];
    buffer[index++] = data[18];
    buffer[index++] = data[19];

    buffer[index++] = data[20];
    buffer[index++] = data[21];
    buffer[index++] = data[22];
    buffer[index++] = data[23];

    size = index;

    buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
    buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
    buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
    buffer[index++] = (byte)(ipSrcAddress & 0xFF);
    buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
    buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
    buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
    buffer[index++] = (byte)(ipDestAddress & 0xFF);

    buffer[index++] = (byte)((size >> 8) & 0xFF);
    buffer[index++] = (byte)(size & 0xFF);

    for(i = 0; i < index; i+=2)
    {
       hi += (buffer[i] & 0xFF);
       lo += (buffer[(i + 1)] & 0xFF);
    }

    lo += ipProtocol;
    tmp = ((lo >> 8) & 0xFF);
    hi += tmp;

    tmp = ((hi >> 8) & 0xFF);
    lo += tmp;

    lo = ((~lo) & 0xFF);
```

```
        hi = ((~hi) & 0xFF);
        tcpChecksum = (short)((hi << 8) | lo);
    }
    public static void SetTcpFlags(int Flags)
    {
        tcpFlags = (byte)Flags;
    }
    public static void SetIpSrcAddress(int Src)
    {
        ipSrcAddress = Src;
    }
    public static void SetIpDestAddress(int Dest)
    {
        ipDestAddress = Dest;
    }
    public static void SetTcpSrcPort(int SrcPort)
    {
        tcpSrcPort = SrcPort;
    }
    public static void SetTcpSeqNumber(long SequenceNumber)
    {
        tcpSeqNumber = SequenceNumber;
    }
    public static void SetTcpAckNumber(long AckNumber)
    {
        tcpAckNumber = AckNumber;
    }
}
```

```
/*****************************************************************************
*
* Description: This module contains the packet for testing the target for
*              TCP checksum validation.
*
* RCS info
* $Id: Test10.java,v 1.3 2001/03/01 01:25:50 dmcgann Exp $
*
* $Log: Test10.java,v $
* Revision 1.3  2001/03/01 01:25:50  dmcgann
* Changed ip checksum to 0x1111, which means do not calculate TCP
* checksum in the injector.
*
* Revision 1.2  2001/03/01 01:05:31  dmcgann
* Added function to compare test results.
*
* Revision 1.1  2001/02/18 01:19:19  dmcgann
* Initial revision
*
*
*
*****************************************************************************/

public class Test10
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =   0x0030;
   private static short   ipId                  =   0x026A;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x1111; /* 1111 means */
   private static long    ipSrcAddress          = 0xC6640019; /* no TCP     */
   private static long    ipDestAddress         = 0xC6640001; /* checksum.  */

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x18;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x68, 0x65, 0x6C, 0x6C, 0x6F, 0x0A, 0x0D, 0};
   private static short   dataLength = 8;

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(data);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int index = 0;
      int j;

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
```





```
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);

      for(j = 0; j < dataLength; j++)
         Buffer[index++] = data[j];

      return(index);
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }

   public static void SetIpDestAddress(int Dest)
   {
      ipDestAddress = Dest;
   }

   public static void SetTcpSrcPort(int SrcPort)
   {
      tcpSrcPort = SrcPort;
   }

   public static void SetTcpSeqNumber(long SequenceNumber)
   {
      tcpSeqNumber = SequenceNumber;
   }

   public static void SetTcpAckNumber(long AckNumber)
   {
```

```
        tcpAckNumber = AckNumber;
    }
}
```

```
/*****************************************************************************
*
* Description: This test sends an IP packet with an invalid IP header
*              length.
*
* RCS info
* $Id: Test11.java,v 1.1 2001/03/23 01:10:06 dmcgann Exp $
*
* $Log: Test11.java,v $
* Revision 1.1  2001/03/23 01:10:06  dmcgann
* Initial revision
*
*
*
*****************************************************************************/
import java.util.*;

public class Test11
{
   private static byte    ipVersAndHeaderLength =     0x43;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =       4B;
   private static short   ipId                  =   0x3014;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;
   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x1B;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x6B, 0x65, 0x6C, 0x6C,
                                  0x6F, 0x0A, 0x0D, 0};

   private static short   dataLength = 1;

   public static void ResetTest()
   {
   }

   public static int NumPacketsInTest()
   {
      return(dataLength);
   }

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(data);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int    index = 0;
      int    j;
      Random rand = new Random();
```





```
      ipId = (short)rand.nextInt(65536);

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);

      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
      Buffer[index++] = data[0];
      Buffer[index++] = data[1];
      Buffer[index++] = data[2];
      Buffer[index++] = data[3];

      Buffer[index++] = data[4];
      Buffer[index++] = data[5];
      Buffer[index++] = data[6];
      Buffer[index++] = data[7];

      return(index);
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }

   public static void SetIpDestAddress(int Dest)
   {
```

```
      ipDestAddress = Dest;
   }

   public static void SetTcpSrcPort(int SrcPort)
   {
      tcpSrcPort = SrcPort;
   }

   public static void SetTcpSeqNumber(long SequenceNumber)
   {
      tcpSeqNumber = SequenceNumber;
   }

   public static void SetTcpAckNumber(long AckNumber)
   {
      tcpAckNumber = AckNumber;
   }
}
```

```
/*****************************************************************************
*
* Description: This function transmits a TCP segment using IP fragmentation
*              with an attempt to overlap fragments.
*
* RCS info
* $Id: Test2.java,v 1.1 2001/03/02 01:50:21 dmcgann Exp $
*
* $Log: Test2.java,v $
* Revision 1.1  2001/03/02 01:50:21  dmcgann
* Initial revision
*
*
*****************************************************************************/
import java.util.*;
import java.lang.*;

public class Test2
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =       24;
   private static short   ipId                  =   0x3014;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;
   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x18;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x68, 0x65, 0x6C, 0x6C,
                                  0x6F, 0x20, 0x20, 0x65,
                                  0x66, 0x67, 0x68, 0x69,
                                  0x70, 0x71, 0x72, 0x73,
                                  0x74, 0x75, 0x76, 0x77,
                                  0x78, 0x79, 0x21, 0x22};

   private static byte[]  data2 = {0x64, 0x64, 0x64, 0x64};

   private static short   dataLength = 7;
   private static short   fragIndex = 0;
   private static boolean checksumCalcd = false;
   private static short   secondTcpChecksum = 0;

   public static void ResetTest()
   {
      Random rand = new Random();

      ipId = (short)rand.nextInt(65535);

      tcpChecksum = 0;
      fragIndex = 0;
      ipFrag = (short)0x2000;
      ipTotalLength = 20 + 8;
      checksumCalcd = false;
   }

   public static int NumPacketsInTest()
   {
      return(dataLength);
   }
```





```
   }

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(data);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int    index = 0;
      int    j;

      if(checksumCalcd == false)
      {
         checksumCalcd = true;
         CalcTcpChecksum();
      }

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      if(fragIndex == 0)
      {
         ipFrag = 0x2001;

         Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
         Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpDestPort & 0xFF);
         Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
         Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
         Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);
      }
      else if(fragIndex == 1)
      {
         ipFrag = 0x2003;

         Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
         Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
         Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
         Buffer[index++] = tcpHeaderLength;
         Buffer[index++] = tcpFlags;
         Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
         Buffer[index++] = (byte)(tcpWinSize & 0xFF);
```

```
    }
    else if(fragIndex == 4)
    {
       Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
       Buffer[index++] = (byte)(tcpChecksum & 0xFF);
       Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
       Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
       Buffer[index++] = data[0];
       Buffer[index++] = data[1];
       Buffer[index++] = data[2];
       Buffer[index++] = data[3];
    }
    else if(fragIndex == 5)
    {
       ipFrag = 0x0005;
       ipTotalLength = 24;

       Buffer[index++] = (byte)((secondTcpChecksum >> 8) & 0xFF);
       Buffer[index++] = (byte)(secondTcpChecksum & 0xFF);
       Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
       Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
       Buffer[index++] = data2[0];
       Buffer[index++] = data2[1];
       Buffer[index++] = data2[2];
       Buffer[index++] = data2[3];
    }
    else if(fragIndex == 2)
    {
       ipFrag = 0x2004;

       Buffer[index++] = data[4];
       Buffer[index++] = data[5];
       Buffer[index++] = data[6];
       Buffer[index++] = data[7];
       Buffer[index++] = data[8];
       Buffer[index++] = data[9];
       Buffer[index++] = data[10];
       Buffer[index++] = data[11];
    }
    else if(fragIndex == 3)
    {
       ipFrag = 0x2002;

       Buffer[index++] = data[12];
       Buffer[index++] = data[13];
       Buffer[index++] = data[14];
       Buffer[index++] = data[15];
       Buffer[index++] = data[16];
       Buffer[index++] = data[17];
       Buffer[index++] = data[18];
       Buffer[index++] = data[19];
    }
    else if(fragIndex == 6)
    {
       Buffer[index++] = data[20];
       Buffer[index++] = data[21];
       Buffer[index++] = data[22];
       Buffer[index++] = data[23];
    }

    fragIndex++;
    return(index);
}

private static void CalcTcpChecksum()
{
    byte[] buffer = new byte[128];
    int    index = 0;
    int    size;
    int    i;
```





```
    int    hi = 0;
    int    lo = 0;
    int    tmp;
    int    hi2;
    int    lo2;

    buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpSrcPort & 0xFF);
    buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpDestPort & 0xFF);
    buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
    buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
    buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpSeqNumber & 0xFF);

    buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
    buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
    buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpAckNumber & 0xFF);
    buffer[index++] = tcpHeaderLength;
    buffer[index++] = tcpFlags;
    buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpWinSize & 0xFF);
    buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpChecksum & 0xFF);
    buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
    buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
    buffer[index++] = data[0];
    buffer[index++] = data[1];
    buffer[index++] = data[2];
    buffer[index++] = data[3];

    buffer[index++] = data[4];
    buffer[index++] = data[5];
    buffer[index++] = data[6];
    buffer[index++] = data[7];
    buffer[index++] = data[8];
    buffer[index++] = data[9];
    buffer[index++] = data[10];
    buffer[index++] = data[11];

    buffer[index++] = data[12];
    buffer[index++] = data[13];
    buffer[index++] = data[14];
    buffer[index++] = data[15];
    buffer[index++] = data[16];
    buffer[index++] = data[17];
    buffer[index++] = data[18];
    buffer[index++] = data[19];

    buffer[index++] = data[20];
    buffer[index++] = data[21];
    buffer[index++] = data[22];
    buffer[index++] = data[23];

    size = index;

    buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
    buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
    buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
    buffer[index++] = (byte)(ipSrcAddress & 0xFF);
    buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
    buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
    buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
    buffer[index++] = (byte)(ipDestAddress & 0xFF);

    buffer[index++] = (byte)((size >> 8) & 0xFF);
    buffer[index++] = (byte)(size & 0xFF);

    for(i = 0; i < index; i+=2)
```

```
    {
       hi += (buffer[i] & 0xFF);
       lo += (buffer[(i + 1)] & 0xFF);
    }

    hi2 = hi;
    lo2 = lo;

    lo += ipProtocol;
    tmp = ((lo >> 8) & 0xFF);
    hi += tmp;

    tmp = ((hi >> 8) & 0xFF);
    lo += tmp;

    lo = ((~lo) & 0xFF);
    hi = ((~hi) & 0xFF);

    tcpChecksum = (short)((hi << 8) | lo);

    /* second checksum */
    lo2 += ipProtocol;

    hi2 -= data[0];
    hi2 -= data[2];
    lo2 -= data[1];
    lo2 -= data[3];

    hi2 += data2[0];
    hi2 += data2[2];
    lo2 += data2[1];
    lo2 += data2[3];

    tmp = ((lo2 >> 8) & 0xFF);
    hi2 += tmp;

    tmp = ((hi2 >> 8) & 0xFF);
    lo2 += tmp;

    lo2 = ((~lo2) & 0xFF);
    hi2 = ((~hi2) & 0xFF);

    secondTcpChecksum = (short)((hi2 << 8) | lo2);
  }

  public static void SetTcpFlags(int Flags)
  {
     tcpFlags = (byte)Flags;
  }

  public static void SetIpSrcAddress(int Src)
  {
     ipSrcAddress = Src;
  }

  public static void SetIpDestAddress(int Dest)
  {
     ipDestAddress = Dest;
  }

  public static void SetTcpSrcPort(int SrcPort)
  {
     tcpSrcPort = SrcPort;
  }

  public static void SetTcpSeqNumber(long SequenceNumber)
  {
     tcpSeqNumber = SequenceNumber;
  }
```





```
  public static void SetTcpAckNumber(long AckNumber)
  {
     tcpAckNumber = AckNumber;
  }
}
```

```
/*****************************************************************************
*
* Description: This test sends a TCP segment with an invalid IP version.
*
* RCS info
* $Id: Test3.java,v 1.1 2001/03/01 00:32:16 dmcgann Exp $
*
* $Log: Test3.java,v $
* Revision 1.1  2001/03/01 00:32:16  dmcgann
* Initial revision
*
*
*****************************************************************************/
import java.util.*;

public class Test3
{
   private static byte    ipVersAndHeaderLength =     0x35;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =       4B;
   private static short   ipId                  =   0x3014;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;
   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x1B;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x6B, 0x65, 0x6C, 0x6C,
                                  0x6F, 0x0A, 0x0D, 0};

   private static short   dataLength = 1;

   public static void ResetTest()
   {
   }

   public static int NumPacketsInTest()
   {
      return(dataLength);
   }

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(data);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int    index = 0;
      int    j;
      Random rand = new Random();

      ipId = (short)rand.nextInt(65536);
```





```
      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);

      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
      Buffer[index++] = data[0];
      Buffer[index++] = data[1];
      Buffer[index++] = data[2];
      Buffer[index++] = data[3];

      Buffer[index++] = data[4];
      Buffer[index++] = data[5];
      Buffer[index++] = data[6];
      Buffer[index++] = data[7];

      return(index);
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }

   public static void SetIpDestAddress(int Dest)
   {
      ipDestAddress = Dest;
   }
```

```
   public static void SetTcpSrcPort(int SrcPort)
   {
      tcpSrcPort = SrcPort;
   }

   public static void SetTcpSeqNumber(long SequenceNumber)
   {
      tcpSeqNumber = SequenceNumber;
   }

   public static void SetTcpAckNumber(long AckNumber)
   {
      tcpAckNumber = AckNumber;
   }
}
```

```
/*****************************************************************************
*
* Description: This test verifies that the target machine checks for
*              an invalid TCP header length. Any header length below
*              5 32-bit words should be dropped, since a minimum of
*              5 double-words are need for the TCP header.
*
* RCS info
* $Id: Test4.java,v 1.3 2001/03/01 01:03:55 dmcgann Exp $
*
* $Log: Test4.java,v $
* Revision 1.3  2001/03/01 01:03:55  dmcgann
* Changed default value of IP checksum to 0.
*
* Revision 1.2  2001/03/01 00:37:40  dmcgann
* Added function for comparing results of test.
*
* Revision 1.1  2001/02/17 22:14:27  dmcgann
* Initial revision
*
*
*****************************************************************************/

public class Test4
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =   0x0030;
   private static short   ipId                  =   0x026A;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;
   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x30;
   private static byte    tcpFlags              =     0x18;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x68, 0x65, 0x6C, 0x6C, 0x6F, 0x0A, 0x0D, 0};
   private static short   dataLength = 8;

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(data);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int index = 0;
      int j;

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
```





```
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);

      for(j = 0; j < dataLength; j++)
         Buffer[index++] = data[j];

      return(index);
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }

   public static void SetIpDestAddress(int Dest)
   {
      ipDestAddress = Dest;
   }

   public static void SetTcpSrcPort(int SrcPort)
   {
      tcpSrcPort = SrcPort;
   }

   public static void SetTcpSeqNumber(long SequenceNumber)
   {
      tcpSeqNumber = SequenceNumber;
   }

   public static void SetTcpAckNumber(long AckNumber)
   {
```

```
        tcpAckNumber = AckNumber;
    }
}
```

```
/*************************************************************************
*
* Description: This test sends a TCP segment with an invalid IP total
*              length.
*
* RCS info
* $Id: Test5.java,v 1.1 2001/03/02 01:48:32 dmcgann Exp $
*
* $Log: Test5.java,v $
* Revision 1.1  2001/03/02 01:48:32  dmcgann
* Initial revision
*
*
*************************************************************************/
import java.util.*;

public class Test5
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =       41;
   private static short   ipId                  =   0x3014;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;

   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x18;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x68, 0x65, 0x6C, 0x6C,
                                  0x6F, 0x0A, 0x0D, 0};

   private static short   dataLength = 1;

   public static void ResetTest()
   {
   }

   public static int NumPacketsInTest()
   {
      return(dataLength);
   }

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(data);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int    index = 0;
      int    j;
      Random rand = new Random();
```





```
      ipId = (short)rand.nextInt(65536);

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);

      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
      Buffer[index++] = data[0];
      Buffer[index++] = data[1];
      Buffer[index++] = data[2];
      Buffer[index++] = data[3];

      Buffer[index++] = data[4];
      Buffer[index++] = data[5];
      Buffer[index++] = data[6];
      Buffer[index++] = data[7];

      return(index);
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }

   public static void SetIpDestAddress(int Dest)
   {
```

```
    ipDestAddress = Dest;
  }

  public static void SetTcpSrcPort(int SrcPort)
  {
    tcpSrcPort = SrcPort;
  }

  public static void SetTcpSeqNumber(long SequenceNumber)
  {
    tcpSeqNumber = SequenceNumber;
  }

  public static void SetTcpAckNumber(long AckNumber)
  {
    tcpAckNumber = AckNumber;
  }
}
```

```
/**************************************************************************
*
* Description: This test sends a TCP segment with an invalid IP checksum.
*
* RCS info
* $Id: Test6.java,v 1.1 2001/03/02 01:46:30 dmcgann Exp $
*
* $Log: Test6.java,v $
* Revision 1.1  2001/03/02 01:46:30  dmcgann
* Initial revision
*
*
**************************************************************************/
import java.util.*;

public class Test6
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =       48;
   private static short   ipId                  =   0x3014;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;

   /* ip checksum means the injector should not calculate */
   /* ip checksum.                                        */
   private static short   ipChecksum            =   0x2222;

   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeadarLength       =     0x50;
   private static byte    tcpFlags              =     0x18;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x68, 0x65, 0x6C, 0x6C,
                                  0x6F, 0x0A, 0x0D, 0};

   private static short   dataLength = 1;

   public static void ResetTest()
   {
   }

   public static int NumPacketsInTest()
   {
      return(dataLength);
   }

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(data);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int    index = 0;
      int    j;
```





```
      Random rand = new Random();

      ipId = (short)rand.nextInt(65536);

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);

      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
      Buffer[index++] = data[0];
      Buffer[index++] = data[1];
      Buffer[index++] = data[2];
      Buffer[index++] = data[3];

      Buffer[index++] = data[4];
      Buffer[index++] = data[5];
      Buffer[index++] = data[6];
      Buffer[index++] = data[7];

      return(index);
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }
```

```
public static void SetIpDestAddress(int Dest)
{
   ipDestAddress = Dest;
}

public static void SetTcpSrcPort(int SrcPort)
{
   tcpSrcPort = SrcPort;
}

public static void SetTcpSeqNumber(long SequenceNumber)
{
   tcpSeqNumber = SequenceNumber;
}

public static void SetTcpAckNumber(long AckNumber)
{
   tcpAckNumber = AckNumber;
}
```

```
/*****************************************************************************
*
* Description: This module contains the packets for testing the target's
*              TCP stream reassembly capability. To test this, 8 1-byte TCP
*              segemnts are sent.
*
* RCS info
* $Id: Test7.java,v 1.3 2001/03/02 01:45:13 dmcgann Exp $
*
* $Log: Test7.java,v $
* Revision 1.3  2001/03/02 01:45:13  dmcgann
* Fixed ip total length field.
*
* Revision 1.2  2001/03/01 00:43:08  dmcgann
* Added function to compare results of test.
*
* Revision 1.1  2001/02/18 16:10:44  dmcgann
* Initial revision
*
*
*****************************************************************************/

public class Test7
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =       41;
   private static short   ipId                  =   0x026A;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;
   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x18;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x68, 0x65, 0x6C, 0x6C, 0x6F, 0x0A, 0x0D, 0};
   private static short   dataLength = 8;
   private static short   dataIndex = 0;

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(data);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static void ResetTest()
   {
      dataIndex = 0;
   }

   public static int NumPacketsInTest()
   {
      return(dataLength);
   }

   public static int GetPacket(byte[] Buffer)
```





```
   {
      int index = 0;
      int j;

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSeqNumber & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);

      /* For test7 we only want a single byte of data */
      /* per packet.                                  */
      if(dataIndex < dataLength)
         Buffer[index++] = data[dataIndex++];
      else
         System.out.println("error: Exceeded number of packets for Test7");

      return(index);
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }

   public static void SetIpDestAddress(int Dest)
   {
      ipDestAddress = Dest;
```

```
/*****************************************************************************
*
* Description: This module contains the packets for testing the target's
*              TCP stream reassembly capability. To test this, 8 1-byte TCP
*              segments are sent. One of the segments will attempts to
*              overwrite a previously sent TCP segment.
*
* RCS info
* $Id: Test8.java,v 1.3 2001/03/02 01:43:48 dmcgann Exp $
*
* $Log: Test8.java,v $
* Revision 1.3  2001/03/02 01:43:48  dmcgann
* Added random function call for ipId.
* Fixed ip total length field.
*
* Revision 1.2  2001/03/01 01:06:17  dmcgann
* Added function to compare test results.
*
* Revision 1.1  2001/02/22 00:41:03  dmcgann
* Initial revision
*
*
*****************************************************************************/
import java.util.*;

public class Test8
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =       41;
   private static short   ipId                  =   0x026A;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;
   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x18;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x68, 0x65, 0x6C, 0x6D,
                                  0x6C, 0x6F, 0x0A, 0x0D, 0};

   private static byte[]  firstData = {0x68, 0x65, 0x6C, 0x6C,
                                       0x6F, 0x0A, 0x0D, 0};

   private static short   dataLength = 9;
   private static short   dataIndex = 0;
   private static int     seqOffset;
   private static boolean firstPass;
   private static boolean baseSeqNumSet;

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(firstData);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }
```





```
   public static void ResetTest()
   {
      Random rand = new Random();

      ipId = (short)rand.nextInt(65535);

      dataIndex = 8;
      seqOffset = 7;
      firstPass = true;
      baseSeqNumSet = false;
   }

   public static int NumPacketsInTest()
   {
      return(dataLength);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int    index = 0;
      int    j;
      long   bufferedSeqNumber = tcpSeqNumber + seqOffset;

      /* Check for first pass, we will be attempting */
      /* to overwrite a previously sent TCP segment. */
      if((seqOffset == 3) && (firstPass == true))
      {
         firstPass = false;
      }
      else
         seqOffset--;

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((bufferedSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((bufferedSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((bufferedSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(bufferedSeqNumber & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);
```

```
      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);

      /* For test8 we only want a single byte of data */
      /* per packet.                                  */
      if(dataIndex >= 0)
         Buffer[index++] = data[dataIndex--];
      else
         System.out.println("error: Exceeded number of packets for Test8");

      return(index);
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }

   public static void SetIpDestAddress(int Dest)
   {
      ipDestAddress = Dest;
   }

   public static void SetTcpSrcPort(int SrcPort)
   {
      tcpSrcPort = SrcPort;
   }

   public static void SetTcpSeqNumber(long SequenceNumber)
   {
      if(baseSeqNumSet == false)
      {
         baseSeqNumSet = true;
         tcpSeqNumber = SequenceNumber;
      }
   }

   public static void SetTcpAckNumber(long AckNumber)
   {
      tcpAckNumber = AckNumber;
   }
}
```

```
/*****************************************************************************
*
* Description: This module contains the packets for testing the target's
*              TCP stream reassembly capability. To test this, 8 1-byte TCP
*              segemnts are sent. One of the segments is 2 bytes in length
*              and will overwrite a previously sent segment.
*
* RCS info
* $Id: Test9.java,v 1.3 2001/03/02 01:37:56 dmcgann Exp $
*
* $Log: Test9.java,v $
* Revision 1.3  2001/03/02 01:37:56  dmcgann
* Added random function call for ipId.
* Fixed IP total length field.
*
* Revision 1.2  2001/03/01 01:05:52  dmcgann
* Added function to compare test results.
*
* Revision 1.1  2001/02/22 00:38:16  dmcgann
* Initial revision
*
*
*****************************************************************************/
import java.util.*;

public class Test9
{
   private static byte    ipVersAndHeaderLength =     0x45;
   private static byte    ipTos                 =     0x00;
   private static short   ipTotalLength         =   0x0030;
   private static short   ipId                  =   0x026A;
   private static short   ipFrag                =   0x0000;
   private static byte    ipTtl                 =     0x40;
   private static byte    ipProtocol            =     0x06;
   private static short   ipChecksum            =   0x0000;
   private static long    ipSrcAddress          = 0xC6640019;
   private static long    ipDestAddress         = 0xC6640001;

   private static int     tcpSrcPort            =   0x0000;
   private static int     tcpDestPort           =   0x3AF0;
   private static long    tcpSeqNumber          = 0x00000000;
   private static long    tcpAckNumber          = 0x00000000;
   private static byte    tcpHeaderLength       =     0x50;
   private static byte    tcpFlags              =     0x18;
   private static short   tcpWinSize            =   0x0400;
   private static short   tcpChecksum           =   0x0000;
   private static short   tcpUrgPtr             =   0x0000;

   private static byte[]  data = {0x68, 0x65, 0x6C, 0x6D,
                                  0x6C, 0x6F, 0x0A, 0x0D, 0};

   private static byte[]  firstData = {0x68, 0x65, 0x6C, 0x6C,
                                       0x6F, 0x0A, 0x0D, 0};

   private static short   dataLength = 8;
   private static short   dataIndex = 0;
   private static int     seqOffset;
   private static boolean baseSeqNumSet;

   public static boolean CompareResults(String RxMesg)
   {
      String SentMesg = new String(firstData);

      if(RxMesg != null)
         return(SentMesg.equals(RxMesg));
      else
         return(false);
   }

   public static void ResetTest()
```





```
   {
      Random rand = new Random();

      ipId = (short)rand.nextInt(65535);

      dataIndex = 8;
      seqOffset = 7;
      baseSeqNumSet = false;
   }

   public static int NumPacketsInTest()
   {
      return(dataLength);
   }

   public static int GetPacket(byte[] Buffer)
   {
      int    index = 0;
      int    j;
      long   bufferedSeqNumber = tcpSeqNumber + seqOffset;

      seqOffset--;

      if(seqOffset == 1)
         ipTotalLength = 42;
      else
         ipTotalLength = 41;

      Buffer[index++] = ipVersAndHeaderLength;
      Buffer[index++] = ipTos;
      Buffer[index++] = (byte)((ipTotalLength >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipTotalLength & 0xFF);
      Buffer[index++] = (byte)((ipId >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipId & 0xFF);
      Buffer[index++] = (byte)((ipFrag >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipFrag & 0xFF);
      Buffer[index++] = ipTtl;
      Buffer[index++] = ipProtocol;
      Buffer[index++] = (byte)((ipChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipChecksum & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipSrcAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipSrcAddress & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 24) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 16) & 0xFF);
      Buffer[index++] = (byte)((ipDestAddress >> 8) & 0xFF);
      Buffer[index++] = (byte)(ipDestAddress & 0xFF);

      Buffer[index++] = (byte)((tcpSrcPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpSrcPort & 0xFF);
      Buffer[index++] = (byte)((tcpDestPort >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpDestPort & 0xFF);
      Buffer[index++] = (byte)((bufferedSeqNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((bufferedSeqNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((bufferedSeqNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(bufferedSeqNumber & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 24) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 16) & 0xFF);
      Buffer[index++] = (byte)((tcpAckNumber >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpAckNumber & 0xFF);
      Buffer[index++] = tcpHeaderLength;
      Buffer[index++] = tcpFlags;
      Buffer[index++] = (byte)((tcpWinSize >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpWinSize & 0xFF);

      Buffer[index++] = (byte)((tcpChecksum >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpChecksum & 0xFF);
      Buffer[index++] = (byte)((tcpUrgPtr >> 8) & 0xFF);
      Buffer[index++] = (byte)(tcpUrgPtr & 0xFF);
```

```
      /* For test7 we only want a single byte of data */
      /* per packet.                                  */
      if(seqOffset != 1)
      {
         if(dataIndex >= 0)
            Buffer[index++] = data[dataIndex--];
         else
            System.out.println("error: Exceeded number of packets for Test9");
      }
      else
      {
         Buffer[index++] = data[2];
         Buffer[index++] = data[3];
         dataIndex = 1;
      }

      return(index);
   }

   public static void SetTcpFlags(int Flags)
   {
      tcpFlags = (byte)Flags;
   }

   public static void SetIpSrcAddress(int Src)
   {
      ipSrcAddress = Src;
   }

   public static void SetIpDestAddress(int Dest)
   {
      ipDestAddress = Dest;
   }

   public static void SetTcpSrcPort(int SrcPort)
   {
      tcpSrcPort = SrcPort;
   }

   public static void SetTcpSeqNumber(long SequenceNumber)
   {
      if(baseSeqNumSet == false)
      {
         baseSeqNumSet = true;
         tcpSeqNumber = SequenceNumber;
      }
   }

   public static void SetTcpAckNumber(long AckNumber)
   {
      tcpAckNumber = AckNumber;
   }
}
```

```
/*****************************************************************************
* RCS info
* $Id: Tester.java,v 1.10 2001/03/30 01:39:03 dmcgann Exp $
*
* $Log: Tester.java,v $
* Revision 1.10  2001/03/30 01:39:03  dmcgann
* Added comments.
* Added test for invalid IP header size.
*
* Revision 1.9  2001/03/11 01:17:31  dmcgann
* Added setting of HostProoperties objects to represent results of
* TCP/IP tests.
*
* Revision 1.8  2001/03/03 16:18:18  dmcgann
* Changed statement for port filtering of messages.
* Edited some of the results statemensts printed.
*
* Revision 1.7  2001/03/02 01:57:16  dmcgann
* Added all tests.
*
* Revision 1.6  2001/02/18 16:11:12  dmcgann
* Added running of Test #7, TCP segment reassembly.
*
* Revision 1.5  2001/02/18 01:22:27  dmcgann
* First rev of tests added.
* Test for TCP checksum and invalid TCP header length.
*
* Revision 1.4  2001/02/03 22:45:18  dmcgann
* Converted address representations to Strings from Integers.
*
* Revision 1.3  2001/02/03 01:07:36  dmcgann
* Sends Mobile.class as a file to remote system.
* Added acknowledgements to synchronize connections to Mobile servers.
*
* Revision 1.2  2001/01/28 00:46:08  dmcgann
* First pass of the serialization of Mobile objects to remote.
*
* Revision 1.1  2001/01/25 01:42:27  dmcgann
* Initial revision
*
*
*****************************************************************************/
import java.net.*;
import java.lang.*;
import java.io.*;
import java.util.*;

class Tester extends Thread
{
   private static  LinkedList   ListOfHostsToTest = new LinkedList();
   private static  String       HostUnderTest = null;
   private static  byte[]       HutEtherAddr = new byte[6];

   private static  Socket       CCsock;
   private static  OutputStream CCoutput;
   private static  InputStream  CCinput;
   private static  byte[]       RemoteRecvdData = new byte[1024];

   private static  Socket       TxSock;
   private static  OutputStream TxOutput;

   private static  LinkedList   PacketsFromHUT = new LinkedList();
   private static  TcpPacket    RxTcpPacket;
   private static  IpPacket     RxIpPacket;
   private static  int          PortToFilter = 0;

   private static  Socket       TestSock;

   private static  HostTcpIpProperties TestResults = null;
```





```
/*****************************************************************************
* Function used by PacketRouter to inform Tester of host to test.
*****************************************************************************/
   public synchronized void TesterAddHostToTest(IpPacket Packet)
   {
      String  ipAddress = Packet.GetDestinationHostAddress();

      if((ipAddress.equals(HostUnderTest) == false) &&
         (ListOfHostsToTest.contains((Object)ipAddress) == false))
      {
         ListOfHostsToTest.add((Object)Packet);

         this.notify();
      }
   }

/*****************************************************************************
* Method for querying which host is currently being tested.
*****************************************************************************/
   public synchronized boolean IsHostUnderTest(String Host)
   {
      return(Host.equals(HostUnderTest));
   }

/*****************************************************************************
* Method to pass IP packets from host under test to Tester.
*****************************************************************************/
   public synchronized void InformIpPacketFromHUT(IpPacket Packet)
   {
      TcpPacket a = Packet.GetTcpPacket();

      if((a.GetSourcePort() == 15088) &&
         (a.GetDestinationPort() == PortToFilter))
      {
         PacketsFromHUT.add((Object)Packet);

         this.notify();

         //System.out.println("debug: recv'd packet from HUT");
      }
   }

/*****************************************************************************
* Thread main loop.
*****************************************************************************/
   public void run()
   {
      boolean success;
      int     i;
      byte[]  buffer = new byte[512];

      EstablishInjectorConnection();

      while(true)
      {
         CheckList();

         System.out.println("Removed from queue ->   + HostUnderTest);

         TestResults = new HostTcpIpProperties(HostUnderTest);

         success = SendMobileCode();

         try
         {
            sleep(1000);
         }
         catch(InterruptedException e)
         {
            System.out.println(e.toString());
```

```
}

success = EstablishCCConnection();

/********** Start test 1 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest1();
   success = DisableTestConnection();
}

/********** Start test 2 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest2();
   success = DisableTestConnection();
}

/********** Start test 3 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest3();
   success = DisableTestConnection();
}

/********** Start test 4 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest4();
   success = DisableTestConnection();
}

/********** Start test 5 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest5();
   success = DisableTestConnection();
}

/********** Start test 6 **********/
```





```
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest6();
   success = DisableTestConnection();
}

/********** Start test 7 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest7();
   success = DisableTestConnection();
}

/********** Start test 8 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest8();
   success = DisableTestConnection();
}

/********** Start test 9 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest9();
   success = DisableTestConnection();
}

/********** Start test 10 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();

if(success == true)
{
   success = RunTest10();
   success = DisableTestConnection();
}

/********** Start test 11 **********/
if(success == true)
   success = InformRemoteStartOfTestCycle();

if(success == true)
   success = EstablishTestConnection();
```

```
         if(success == true)
         {
            success = RunTest11();
            success = DisableTestConnection();
         }

         success = InformRemoteTerminationOfTestCycle();

         success = DisableCCCConnection();

         TestResults.Print();

         if(success == true)
         {
            IpProcessor ipProc = new IpProcessor(TestResults);
            ipProc.start();
            PacketRouter.AddIpProcessor(ipProc);
         }
      }
   }

/*****************************************************************************
* Function to check for Host to test.
*****************************************************************************/
   private synchronized void CheckList()
   {
      while(ListOfHostsToTest.isEmpty() == true)
      {
         try
         {
            wait();
         }
         catch(InterruptedException e)
         {
           System.out.println(e.toString());
         }
      }

      IpPacket packet = (IpPacket)ListOfHostsToTest.removeFirst();
      HostUnderTest = packet.GetDestinationHostAddress();
      HutEtherAddr = packet.GetDestinationEthernetAddrBytes();
   }

/*****************************************************************************
* Function for waiting for responses from host under test. Has a 5 second
* timeout.
*****************************************************************************/
   private synchronized boolean WaitForTraffic()
   {
      boolean timedOut = false;

      while((PacketsFromHUT.isEmpty() == true) && (timedOut == false))
      {
         try
         {
            wait(5000);

            if(PacketsFromHUT.isEmpty() == true)
               timedOut = true;
         }
         catch(InterruptedException e)
         {
           System.out.println(e.toString());
         }
      }

      if(timedOut == false)
      {
         //System.out.println("debug: pulling packet");
```





```
         RxIpPacket = (IpPacket)PacketsFromHUT.removeFirst();
         RxTcpPacket = RxIpPacket.GetTcpPacket();
      }
      else
         System.out.println("debug: wait for traffic timed out");


      return(!timedOut);
   }

/*****************************************************************************
* This function sends the Client Data Sink application to the Mobile Code
* Platform.
*****************************************************************************/
   private boolean SendMobileCode()
   {
      boolean success = false;
      int     result;

      try
      {
         int amount;
         byte[] buffer = new byte[127];
         byte   bamount = 0;

         InetAddress localAddr = InetAddress.getByName(HostUnderTest);

         Socket sock = new Socket(localAddr, 17770);

         OutputStream output = sock.getOutputStream();

         InputStream input = sock.getInputStream();

         FileInputStream fInput = new FileInputStream("Mobile.class");

         amount = fInput.read(buffer);

         while(amount > 0)
         {
            bamount = (byte)amount;
            output.write(bamount);

            output.write(buffer, 0, amount);

            amount = fInput.read(buffer);
         }

         bamount = 0;
         output.write(bamount);

         output.flush();

         /* wait for acknowledgement that class has been received. */
         result = input.read();

         if(result == 1)
         {
            System.out.println("All systems go.");
            success = true;
         }
         else
         {
            System.out.println("We have a failure to communicate.");
         }


         fInput.close();
         output.close();
         input.close();
         sock.close();
```

```
        }
        catch(IOException e)
        {
           System.out.println(e.toString());
        }

        return(success);
     }

  /*************************************************************************
  * Function attempts to connect to Client Data Sink's Command & Control
  * Server which it starts up immediately once it starts executing.
  *************************************************************************/
     private boolean EstablishCCConnection()
     {
        boolean success = false;

        try
        {
           InetAddress localAddr = InetAddress.getByName(HostUnderTest);

           CCsock = new Socket(localAddr, 15087);

           System.out.println("Connected:    + CCsock.toString());

           CCoutput = CCsock.getOutputStream();
           CCinput = CCsock.getInputStream();

           success = true;
        }
        catch(IOException e)
        {
           System.out.println(e.toString());
        }

        return(success);
     }

  /*************************************************************************
  * Terminates Command & Control connection between Tester and Client Data
  * Sink.
  *************************************************************************/
     private boolean DisableCCConnection()
     {
        try
        {
           CCinput.close();
           CCoutput.close();
           CCsock.close();
        }
        catch(IOException e)
        {
           System.out.println(e.toString());
        }

        return(true);
     }

  /*************************************************************************
  * This method attempts to connect to the Client Data Sinks testing server.
  *************************************************************************/
     private boolean EstablishTestConnection()
     {
        boolean success = false;

        try
        {
           InetAddress localAddr = InetAddress.getByName(HostUnderTest);

           TestSock = new Socket(localAddr, 15088);
```





```
           PortToFilter = TestSock.getLocalPort();

           System.out.println("Connected:    + TestSock.toString());

           success = true;
        }
        catch(IOException e)
        {
           System.out.println(e.toString());
        }

        return(success);
     }

  /*************************************************************************
  * This method disables the test interface between the Tester and the Client
  * Data Sink.
  *************************************************************************/
     private boolean DisableTestConnection()
     {
        try
        {
           TestSock.close();
        }
        catch(IOException e)
        {
           System.out.println(e.toString());
        }

        return(true);
     }

  /*************************************************************************
  * Commands the Client Data Sink via the CC interface that a test scenario
  * is starting and that a connection attempt on the testing interface is
  * forthcoming.
  *************************************************************************/
     private boolean InformRemoteStartOfTestCycle()
     {
        boolean success = false;
        int     ack;

        try
        {
           CCoutput.write(2);

           ack = CCinput.read();

           if(ack == 1)
           {
              success = true;
           }
           else
              success = false;
        }
        catch(IOException e)
        {
           System.out.println(e.toString());
        }

        return(success);
     }

  /*************************************************************************
  * Method for requesting results of test scenario for Client Data Sink.
  *************************************************************************/
     private int InformRemoteRequestData()
     {
        boolean success = false;
```

```
      int      ack;
      int      length = 0;
      int      index = 0;
      int      amount;

      try
      {
         CCoutput.write(3);

         ack = CCinput.read();

         if(ack == 1)
         {
            success = true;

            length = CCinput.read();

            //System.out.println("request ack =    + length);

            index = 0;

            do
            {
               amount = CCinput.read(RemoteRecvdData, index, length);

               if(amount >= 0)
               {
                  index += amount;
                  length -= amount;
               }

            } while((amount > 0) && (length > 0));
         }
         else
            success = false;
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }

      return(index);
   }

/************************************************************************
* Method for informing Client Data Sink that all tests are complete and that
* it can terminate execution.
************************************************************************/
   private boolean InformRemoteTerminationOfTestCycle()
   {
      boolean success = false;
      int      ack;

      try
      {
         CCoutput.write(0);

         ack = CCinput.read();

         if(ack == 1)
            success = true;
         else
            success = false;
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }

      return(success);
```





```
   }

/************************************************************************
* This function establish a local connection to the Network Packet Injector.
************************************************************************/
   private boolean EstablishInjectorConnection()
   {
      boolean success = false;

      try
      {
         InetAddress localAddr = InetAddress.getLocalHost();

         TxSock = new Socket(localAddr, 15089);

         System.out.println("Connected:    + TxSock.toString());

         TxOutput = TxSock.getOutputStream();

         success = true;
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }

      return(success);
   }

/************************************************************************
* This function writes a packet to the Network Packet Injector.
************************************************************************/
   private boolean WritePacketToInjector(byte[] Packet, int PacketLength)
   {
      byte[] length = new byte[4];

      PacketLength += 6;

      ByteUtils.Word16BitsToBytes(length, PacketLength);

      try
      {
         TxOutput.write(length, 0, 4);

         TxOutput.write(HutEtherAddr, 0, 6);

         //System.out.println("Sent length " + (PacketLength - 6) +
         //    " to injector");

         TxOutput.write(Packet, 0, (PacketLength - 6));
      }
      catch(IOException e)
      {
         System.out.println(e.toString());
      }

      return(true);
   }

/************************************************************************
* This method executes the invalid TCP header length test.
************************************************************************/
   private boolean RunTest4()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int      packetLength;
      int      rxMesgLength;
      int      attempts = 0;
```

```
      //System.out.println("Waiting for traffic...");

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      //System.out.println("Recvd traffic.");

      Test4.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Test4.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Test4.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Test4.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Test4.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Test4.GetPacket(buffer);

      /* wait for acknowledgment */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false) &&
              (++attempts < 3));

      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: test4 rx mesg -> " + rxMesg);

      if(Test4.CompareResults(rxMesg) == true)
      {
         System.out.println("Target assumed min TCP header size");
         TestResults.InvalidTcpHeaderSize = 1;
      }
      else if(rxMesgLength == 0)
      {
         System.out.println("Target rejected invalid TCP header length packet");
         TestResults.InvalidTcpHeaderSize = 0;
      }
      else
      {
         System.out.println("Target failed to validate TCP header length");
         TestResults.InvalidTcpHeaderSize = 2;
      }

      return(success);
   }
```





```
/*************************************************************************
* This method executes the invalid TCP checksum test.
*************************************************************************/
   private boolean RunTest10()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
      int     attempts = 0;

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      Test10.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Test10.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Test10.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Test10.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Test10.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Test10.GetPacket(buffer);

      /* wait for acknowledgment - since we are sending a */
      /* packet with an invalid checksum, we must timeout */
      /* on attempts to send packet.                      */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

         attempts++;

      } while((success == true) &&
              (WaitForTraffic() == false) &&
              (attempts <= 3));

      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: test10 rx mesg ->   + rxMesg);

      if(Test10.CompareResults(rxMesg) == false)
      {
         System.out.println("Target validated TCP checksum");
         TestResults.AcceptInvalidTcpChecksum = false;
      }
      else
      {
```

```
         System.out.println("Target did not validated TCP checksum");
         TestResults.AcceptInvalidTcpChecksum = true;
      }

      return(success);
   }

/*****************************************************************************
* This method executes the 1-byte TCP segment test.
*****************************************************************************/
   private boolean RunTest7()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
      int     i;

      /* Since we are sending a series of packets, we */
      /* must reset the test module.                  */
      Test7.ResetTest();

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      for(i = 0; i < Test7.NumPacketsInTest(); i++)
      {
         Test7.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
         Test7.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
         Test7.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

         Test7.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
         Test7.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

         packetLength = Test7.GetPacket(buffer);

         /* wait for acknowledgment - since we are sending a */
         /* packet with an invalid checksum, we must timeout */
         /* on attempts tc send packet.                      */
         do
         {
            success = WritePacketToInjector(buffer, packetLength);

         } while((success == true) && (WaitForTraffic() == false));
      }

      /* write FIN packet to terminate connection. */
      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();
```





```
      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: test7 rx mesg ->   + rxMesg);

      if(Test7.CompareResults(rxMesg) == true)
         System.out.println("Target reassembled TCP stream correctly");
      else
         System.out.println("Target did not reassemble TCP stream");

      return(success);
   }

/*****************************************************************************
* This method executes the TCP segment rewrite test.
*****************************************************************************/
   private boolean RunTest8()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
      int     i;

      /* Since we are sending a series of packets, we */
      /* must reset the test module.                  */
      Test8.ResetTest();

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      for(i = 0; i < Test8.NumPacketsInTest(); i++)
      {
         Test8.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
         Test8.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
         Test8.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

         Test8.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
         Test8.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

         packetLength = Test8.GetPacket(buffer);

         /* wait for acknowledgment - since we are sending a */
         /* packet with an invalid checksum, we must timeout */
         /* on attempts to send packet.                      */
         do
         {
            success = WritePacketToInjector(buffer, packetLength);

         } while((success == true) && (WaitForTraffic() == false));
      }

      /* write FIN packet to terminate connection. */
      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
```

```
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: test8 rx mesg -> " + rxMesg);

      if(Test8.CompareResults(rxMesg) == true)
      {
         System.out.println("Target did not allow TCP rewrite");
         TestResults.AllowTcpSeqRewrite = false;
      }
      else
      {
         System.out.println("Target allowed TCP rewrite");
         TestResults.AllowTcpSeqRewrite = true;
      }

      return(success);
   }

/*****************************************************************************
* This method executes the TCP segment overlap test.
*****************************************************************************/
   private boolean RunTest9()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
      int     i;

      /* Since we are sending a series of packets, we */
      /* must reset the test module.                  */
      Test9.ResetTest();

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      for(i = 0; i < Test9.NumPacketsInTest(); i++)
      {
         Test9.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
         Test9.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
         Test9.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

         Test9.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
         Test9.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

         packetLength = Test9.GetPacket(buffer);

         /* wait for acknowledgment - since we are sending a */
         /* packet with an invalid checksum, we must timeout */
         /* on attempts to send packet.                      */
         do
         {
            success = WritePacketToInjector(buffer, packetLength);

         } while((success == true) && (WaitForTraffic() == false));
      }

      /* write FIN packet to terminate connection. */
      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());
```





```
      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: test9 rx mesg ->   + rxMesg);

      if(Test9.CompareResults(rxMesg) == true)
      {
         System.out.println("Target did not allow TCP overlap rewrite");
         TestResults.AllowTcpSeqOverlap = false;
      }
      else
      {
         System.out.println("Target did allowed TCP overlap rewrite");
         TestResults.AllowTcpSeqOverlap = true;
      }

      return(success);
   }

/*****************************************************************************
* This function executes the 8-byte IP fragmentation test.
*****************************************************************************/
   private boolean RunTest1()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
      int     i;
      int     attempts = 0;

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      do
      {
         /* Since we are sending a series of packets, we */
         /* must reset the test module.                  */
         Test1.ResetTest();

         for(i = 0; i < Test1.NumPacketsInTest(); i++)
         {

            Test1.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
            Test1.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
            Test1.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

            Test1.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
            Test1.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

            packetLength = Test1.GetPacket(buffer);

            success = WritePacketToInjector(buffer, packetLength);

         }
```

```
      } while((success == true) && (++attempts < 4) && (WaitForTraffic() == false)

      /* write FIN packet to terminate connection. */
      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: test1 rx mesg -> ' + rxMesg);

      if(Test1.CompareResults(rxMesg) == true)
      {
         System.out.println("Target reassembled IP fragments correctly");
         TestResults.AcceptsIpFrags = true;
      }
      else
      {
         System.out.println("Target did not reassemble IP fragments");
         TestResults.AcceptsIpFrags = false;
      }

      return(success);
   }

/*************************************************************************
* This method executes the invlaid IP version test.
*************************************************************************/
   private boolean RunTest3()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
      int     i;
      int     attempts = 0;

      /* Since we are sending a series of packets, we */
      /* must reset the test module.                  */
      Test3.ResetTest();

      //System.out.println("Waiting for traffic...");

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      //System.out.println("Recvd traffic");

      for(i = 0; i < Test3.NumPacketsInTest(); i++)
      {
         Test3.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
```





```
         Test3.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
         Test3.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

         Test3.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
         Test3.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

         packetLength = Test3.GetPacket(buffer);

         /* wait for acknowledgment - since we are sending a */
         /* packet with an invalid checksum, we must timeout */
         /* on attempts to send packet.                      */
         do
         {
            success = WritePacketToInjector(buffer, packetLength);

         } while((success == true) && (++attempts <= 3) &&
                 (WaitForTraffic() == false));
      }

      /* write FIN packet to terminate connection. */
      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: Test3 rx mesg ->   + rxMesg);

      if(Test3.CompareResults(rxMesg) == true)
      {
         System.out.println("Target did not validate IP version field");
         TestResults.AcceptInvalidIpVersion = true;
      }
      else
      {
         System.out.println("Target did validate IP version");
         TestResults.AcceptInvalidIpVersion = false;
      }

      return(success);
   }

/*************************************************************************
* This method executes the invalid IP checksum test.
*************************************************************************/
   private boolean RunTest6()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
```

```
      int     attempts = 0;

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      Test6.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Test6.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Test6.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Test6.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Test6.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Test6.GetPacket(buffer);

      /* wait for acknowledgment - since we are sending a */
      /* packet with an invalid checksum, we must timeout */
      /* on attempts to send packet.                      */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

         attempts++;

      } while((success == true) &&
              (WaitForTraffic() == false) &&
              (attempts <= 3));

      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: test6 rx mesg ->   + rxMesg);

      if(Test6.CompareResults(rxMesg) == false)
      {
         System.out.println("Target validated IP checksum");
         TestResults.AcceptInvalidIpChecksum = false;
      }
      else
      {
         System.out.println("Target did not validated IP checksum");
         TestResults.AcceptInvalidIpChecksum = true;
      }

      return(success);
   }

/*****************************************************************************
* This method executes the invalid IP total packet length test.
```





```
*****************************************************************************/
   private boolean RunTest5()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
      int     attempts = 0;

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      Test5.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Test5.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Test5.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Test5.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Test5.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Test5.GetPacket(buffer);

      /* wait for acknowledgment - since we are sending a */
      /* packet with an invalid checksum, we must timeout */
      /* on attempts to send packet.                      */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

         attempts++;

      } while((success == true) &&
              (WaitForTraffic() == false) &&
              (attempts <= 3));

      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: test5 rx mesg -> " + rxMesg);

      if(rxMesgLength == 0)
      {
         System.out.println("Target rejected abnormal IP total length");
         TestResults.IncorrectIpTotalLength = 0;
      }
      else if(Test5.CompareResults(rxMesg) == false)
      {
         System.out.println("Target used IP total length fields");
         TestResults.IncorrectIpTotalLength = 1;
```

```
      }
      else
      {
         System.out.println("Target did not use IP total length field");
         TestResults.IncorrectIpTotalLength = 2;
      }

      return(success);
   }

/*************************************************************************
* This function executes the IP fragmentation overlap test.
*************************************************************************/
   private boolean RunTest2()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
      int     i;
      int     attempts = 0;

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      do
      {
         /* Since we are sending a series of packets, we */
         /* must reset the test module.                  */
         Test2.ResetTest();

         for(i = 0; i < Test2.NumPacketsInTest(); i++)
         {

            Test2.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
            Test2.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
            Test2.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

            Test2.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
            Test2.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

            packetLength = Test2.GetPacket(buffer);

            success = WritePacketToInjector(buffer, packetLength);

         }
      } while((success == true) && (++attempts < 4) && (WaitForTraffic() == false));

      /* write FIN packet to terminate connection. */
      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();
```





```
      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: test2 rx mesg ->   + rxMesg);

      if(Test2.CompareResults(rxMesg) == true)
      {
         System.out.println("Target did not allow IP frag rewrites");
         TestResults.AllowsIpFragRewrite = false;
      }
      else
      {
         System.out.println("Target allowed IP frag rewrites");
         TestResults.AllowsIpFragRewrite = true;
      }

      return(success);
   }

/*************************************************************************
* This function executes the invalid IP header size test.
*************************************************************************/
   private boolean RunTest11()
   {
      byte[]  buffer = new byte[512];
      boolean success = false;
      int     packetLength;
      int     rxMesgLength;
      int     i;
      int     attempts = 0;

      /* Since we are sending a series of packets, we */
      /* must reset the test module.                  */
      Test11.ResetTest();

      //System.out.println("Waiting for traffic...");

      /* retrieve SYNC ack to learn seq and ack numbers */
      success = WaitForTraffic();

      //System.out.println("Recvd traffic");

      for(i = 0; i < Test11.NumPacketsInTest(); i++)
      {
         Test11.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
         Test11.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
         Test11.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());

         Test11.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
         Test11.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

         packetLength = Test11.GetPacket(buffer);

         /* wait for acknowledgment - since we are sending a */
         /* packet with an invalid checksum, we must timeout */
         /* on attempts to send packet.                      */
         do
         {
            success = WritePacketToInjector(buffer, packetLength);

         } while((success == true) && (++attempts <= 3) &&
                 (WaitForTraffic() == false));
      }

      /* write FIN packet to terminate connection. */
      Disconnect.SetTcpSeqNumber(RxTcpPacket.GetAckNumber());
      Disconnect.SetTcpAckNumber(RxTcpPacket.GetSequenceNumber());
      Disconnect.SetTcpSrcPort(RxTcpPacket.GetDestinationPort());
```

```
      Disconnect.SetIpSrcAddress(RxIpPacket.GetDestinationHostAddressInt());
      Disconnect.SetIpDestAddress(RxIpPacket.GetSourceHostAddressInt());

      packetLength = Disconnect.GetPacket(buffer);

      /* wait for FIN ack */
      do
      {
         success = WritePacketToInjector(buffer, packetLength);

      } while((success == true) && (WaitForTraffic() == false));

      /* filter reception of any packets since test is complete */
      /* and connection should be closed.                       */
      PortToFilter = 0;
      PacketsFromHUT.clear();

      rxMesgLength = InformRemoteRequestData();

      String rxMesg = new String(RemoteRecvdData, 0, rxMesgLength);

      System.out.println("debug: Test11 rx mesg ->   + rxMesg);

      if(Test11.CompareResults(rxMesg) == true)
      {
         System.out.println("Target did not validate IP header length");
         TestResults.AcceptInvalidIpHeaderLength = true;
      }
      else
      {
         System.out.println("Target did validate IP header length");
         TestResults.AcceptInvalidIpHeaderLength = false;
      }

      return(success);
   }
}
```